A Constraint Based Approach For Building Operationally Responsive Satellites

THESIS

Mesut Özkan Kahraman, 1st Lt, TUAF

AFIT/GSS/ENY/08-S02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GSS/ENY/08-S02

A Constraint Based Approach For Building Operationally Responsive Satellites

THESIS

Presented to the Faculty

Department of Aeronautics and Astronautics Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science (Space Systems)

Mesut Özkan Kahraman, B.S.

1st Lt, TUAF

September 2008

AFIT/GSS/ENY/08-S02

A Constraint Based Approach For Building Operationally Responsive Satellites

Mesut Özkan Kahraman, B.S.

1st Lt, TUAF

Approved:

[illegible]	15 Aug 08
Maj Eric D. Swenson, PhD (Chairman)	date
[illegible]	15 Aug 08
Dr. Richard G. Cobb (Member)	date
[illegible]	15 Aug 2008
Dr. Jonathan T. Black (Member)	date

AFIT/GSS/ENY/08-S02

Abstract

The Operational Responsive Space (ORS) program requires flexible and responsive satellites to meet user's needs. Traditional satellite design methods are typically iterative processes that optimize individual components, subsystems, and ultimately the entire satellite. This study focuses on developing a new approach for creating Responsive Satellites (RS) from Plug-and-Play (PnP) components. The aim is to create an approach that quickly evaluates a wide variety of possible satellite configurations and identify the best configurations that meet the user's needs and constraints.

Satellite configurations are created by matching locations on the satellite structure with PnP components. Various constraints are derived from the user's inputs at different levels of the configuration process. As the user provides more information related to PnP satellite, additional constraints can be applied to reduce the number of PnP satellite configurations resulting in manageable numbers or even zero configurations.

In this research, we found that applying constraints whenever it is applicable results in eliminating invalid configurations. Each satellite configuration is saved to a database, if the user desires, a sorted list can help the user find the lowest mass and least expensive satellite that meets their requirements. Configurations can also be eliminated when respective properties are very close to each other which will reduce the number of satellite configurations from which the user can select. A goal of this research effort is to help users assess basic concept feasibility from several key aspects in a short period of time. Finally, more specialized and computationally demanding estimation tools could be called from this approach to perform further analysis, such as thermal, vibration, or structural, to compare and contrast performance characteristics of various satellite configurations.

Acknowledgements

I would like to convey my sincere thanks to Maj Eric Swenson, my research advisor. His insightful guidance along the way was instrumental to the success of this research effort. I would also like to thank Dr Richard Cobb and Dr Jonathan Black for their help and partaking in this effort as committee members.

Mesut Özkan Kahraman

Table of Contents

Page

List of Figures

List of Tables

List of Abbreviations

Abbreviation		Page
ORS	Operational Responsive Space	iv
RS	Responsive Satellite	iv
PnP	Plug-and-Play	iv
DoD	Department of Defense	1
JFC	Joint Forces Commanders	1
AFRL	Air Force Research Laboratory	1
DARPA	Defense Advanced Research Projects Agency	1
USAF	US Air Force	1
COTS	Commercial Off-The-Shelf	2
SMAD	Spacecraft Mission and Analysis and Design	8
AI	Artificial Intelligence	8
SINDA	Systems Improved Numerical Differencing Analyzer	8
Nastran	NASA Structural Analysis	8
SDT	Structural Dynamics Toolbox	8
FE	Finite Element	8
FEA	Finite Element Analysis	8
RST	Responsive Space Testbed	9
HWIL	Hardware-In-Loop	9
DUT	Device Under Test	9
GUI	Graphical User Interface	9
I/O	Input/Output	10
DoF	Degree of Freedom	10
ECI	Earth Centered Inertial	10
CAD	Computer Aided Design	11
3D	Three-Dimensional	20

A Constraint Based Approach For

Building Operationally Responsive Satellites

I. Introduction

As a result of the need for responsiveness in space assets, the trend in satellite design is progressing toward producing smaller more capable satellites. The Department of Defense (DoD) is developing an Operationally Responsive Space (ORS) capability to rapidly put a satellite into orbit shortly after the request is made by Joint Forces Commanders (JFC) or other users in field. In support of this capability, the Air Force Research Laboratory (AFRL), Defense Advanced Research Projects Agency (DARPA), and other DoD organizations have invested heavily in technologies that allow satellites to be designed and assembled using more modular approaches. This ORS capability should result in a significant contribution to the current space capabilities in the terms of military and civilian applications such as replacing failed satellites quickly, augmenting/surging existing capabilities, filling unanticipated gaps in capabilities, and enhancing survivability and deterrence.

1.1 Background

Typical satellite development programs can take more than a decade and even small satellites - those less than 1,000 lbs - can take up to 5 to 7 years to develop [21]. In traditional satellite design, developers may encounter many difficulties in selecting components at the planning stage due to the fact that it is not easy to carry out precise estimations of cost, weight, and power. Interdependencies among the subsystems force developers to use an iterative process of modifying the design of individual components and subsystem until all requirements are met. Each iteration changes the weight and power budget resulting in some components iteratively replaced with other components. Growing costs and schedules due to challenging mission requirements have made space increasingly unresponsive. The US Air Force (USAF) has developed a Responsive Space (RS) program to reduce the time required to design, build, test, and field small satellites. One goal of the RS program is to assemble a satellite in only 6 days [11]. To achieve this goal, all components would have to be readily available and quickly identified for integration and test. To minimize the integration and test process, pretested, modular, and standardized components would have to be used. Developing low-cost, highly reliable components that are Plug-and-Play (PnP), similar to the modular standards used in today's PCs, will simplify this process to the point that a wide variety of satellite configurations can be assembled from a standardized set of components.

1.2 Plug-And-Play Satellite Development

The ORS approach consists of three tiers [1]:

1. Utilize existing on-orbit capabilities to provide the required space-based capability within days from establishing the need,
2. Utilize field-ready or nearly field-ready capabilities to satisfy warfighter needs within days-to-weeks of establishing the need, and
3. Develop an entirely new capability within a year.

Traditional methods used to build small satellites are inadequate for meeting the Tier 3 objectives, therefore, a new method of designing satellites using PnP components is required. The PnP components allow designers to create networks by simply plugging components together (self-organization) and to connect to other devices through use of electronic datasheets embedded in components (self-description).

The ORS program requires the use of commercial off-the-shelf (COTS) components to reduce both the complexity and cost of the satellite which ultimately increases the reliability of the satellites [16]. Currently, there is little standardization among software and hardware developers for PnP components. However, there have been many studies focused on developing PnP components [1], [8], [12], [15]. There

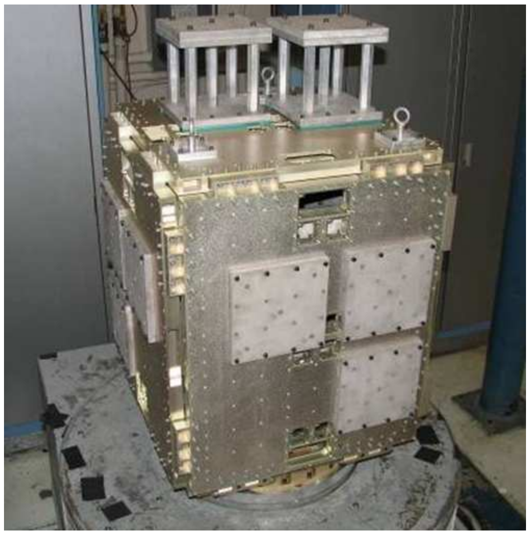


Figure 1.1: PnPSat structural engineering model [1]

has also been a concerted effort by the ORS office and AFRL to create PnP standards resulting in the first PnP Satellite which is expected to launch in the 2008/2009 timeframe. Figure 1.1 shows the structural engineering model of PnPSat ready for the test for Z-axis characterization and loading.

1.3 Problem Description

Quickly determining which components and overall configuration best satisfy the user's constraints, like cost, weight, size, or life span, is very difficult because of complexity of satellite subsystems, numerous of available component choices, limited budgets, and high reliability requirement of space assets. While configuring a satellite system, decisions made for a subsystem often influence other subsystems resulting in changes of constraints and forcing an iteration in the design process. For example, if there are four different battery packs available in a warehouse that satisfy all requirements and constraints, the number of plausible combinations increases significantly if there are also many locations in which the battery packs could be located. The goal is to create a software tool to first determining which configurations meet the user's needs and constraints and then estimate a wide variety of performance specifications based on each satellite configurations, like pointing accuracy or slew rate. Therefore, it would be possible for a single engineer to assess the basic concept feasibility from several key aspects in a short period of time. More specialized and computationally demanding estimation tools could be called after the satellite configurations have been determined to perform further analyses, such as thermal, vibration, or structural. This would allow users to compare and contrast performance characteristics of various satellite configurations.

The end goal of this study is to create, analyze, and test an approach for this rapid satellite configuration problem. This approach will rapidly determine the configuration space from which cost and performance estimates of the possible responsive satellite systems (configurations) are computed so the end user can rapidly select the best satellite system that suits their needs.

1.4 Motivation

The notion of assembling complete systems out of prefabricated parts is a common approach in final product manufacturing. Using COTS components leads to the creation of automated, prompt, and economical products because standardized components were developed to meet the developers' requirements. The properties of components are typically provided by the manufacturers that include all functional and non-functional attributes of the COTS components which allows end product developers to assess products before selection and assembly.

Similarly, manufacturing satellites to meet the short timeline requirements of ORS requires PnP components to be used. Evaluating all possible configurations in a design space full of PnP components with different attributes and interfaces is a very challenging problem. Literally, tens of millions of valid satellite configurations could be built from a few hundred components.

The future integration environment of PnP satellites will likely be composed of thousands of PnP components and their respective software components. Current satellite design approaches rely heavily on manual configuration methods in which developers are extremely involved in each step. Moreover, rapidly configuring satellites composed of thousands of components is not possible with current approaches.

Typically, satellite systems are very complex and estimating their performance characteristics is not a simple process because of the interconnection and dependencies between components. Component meeting a single requirement by itself , may not meet the requirements when assembled together with other components. Thus, there is need of a software approach that helps developers analyze all possible satellite configurations that can be created from a repository of PnP components and reduce the number of the configurations to manageable numbers by applying constraints during the satellite development process.

1.5 Outline of Thesis

Chapter II, Rapid Satellite Integration Studies, is divided in three sections: Section 2.1, Satellite Design Software Tools, and Section 2.2, Satellite Design Approaches review and analyze existing satellite design concepts and developments that are related to this research effort. Section 2.3, Plug-and-Play Developments, briefly describes developments in the Plug-and-Play technology to achieve the goal of building RS satellite in weeks. Chapter III, Optimizing the Process of Building Responsive Satellites, presents the key methodologies used in this research effort. Section 3.1, Object Oriented Design, explains some of the key features of object-oriented programming directly related to this approach. Section 3.2, Gaussian Probability Distribution Function (PDF), provides a brief explanation of the Gaussian PDF that is used to generate component property values such as mass, cost, etc. Section 3.3, Multiplication Principle, Combinations, and Permutations, explains the principles of multiplication, permutations and combinations applied to this component configuration problem. Section 3.4, Overview of the Method, gives a brief overview of the approach created in this research effort. Some key elements of this approach like Virtual Warehouse, 3D Environment, and Graphical User Interfaces are discussed in Sections 3.5, 3.6, and 3.7, respectively. Section 3.8, Methods Developed to Create PnP Satellites, covers the key steps of the approach to generate valid PnP satellite configurations from known components. Chapter IV, Results and Analysis, discusses several test scenarios created to analyze the approach developed in this research. Finally, Chapter V, Conclusions and Future Work, describes the key conclusions derived from this research effort and makes suggestions for future research.

II. Rapid Satellite Integration Studies

The purpose of this chapter is to review and analyze existing satellite design approaches and recent PnP developments that are related to this research effort. This chapter is divided in three sections: Satellite Design Software Tools, Satellite Design Approaches, and PnP Developments. The first two sections are focused on describing existing approaches for designing the "best feasible" ways for building satellites in a shorter time. The last section briefly describes developments in the PnP technology to achieve the goal of building RS satellite in weeks. The title for every subsection is formatted such that Organization/Company/Author is listed first and Research Area/Software Tool is listed second.

2.1 Satellite Design Software Tools

Engineers in the satellite design field are concerned with creating simpler and more accurate models of their satellites to make the satellite design process faster and more accurate. Although there are numerous freeware and commercial software modeling tools available to perform orbit/constellation, communication and attitude analysis, only a small number of programs exist that assist users in the process of rapidly building and designing finalized designs of satellite. The following section describes the variety of software applications developed to help designers shorten design times for either complete or partial satellites.

2.1.1 Princeton Synergetics Inc. and User Systems - Spacecraft Systems Design and Simulation Environment (SSDSE). SSDSE is a software tool created to assist the user in designing, sizing, and simulating satellite systems [6]. SSDSE consists of four main modules: System Design and Sizing, System Resource Simulation, Component Level Database, and Cost Estimation. The System Design and Sizing Module creates subsystem characteristics using the performance and mission requirements provided by the user. The Simulation Module utilizes the subsystem characteristics to simulate resource interaction as a function of time to identify problems or conflicts with subsystems of the satellite. After eliminating the problems identified in the simulation by adjusting the subsystem characteristics, the user can choose candidate components from a database to form a satellite configuration. The System Design and Sizing Module was developed in Microsoft Excel integrated with Fortran compiled codes to enhance portability of the module.

SSDSE has payload, launch vehicle, and bus subsystem databases for the user to access. One advantage of this program is that the user is capable of changing the attributes of the subsystems and components in the database anytime during the configuration process. An iterative approach, very similar to the approach in Spacecraft Mission and Analysis and Design (SMAD) [22] which will be described in Subsection 2.2.1, is used to estimate the mass and power interactions to estimate the attributes of each subsystem. The Cost Estimation Module uses historical mass/cost data to estimate the total cost of the satellite. The Cost Estimation Module may not lead to correct estimations because there might be new technology available at a lower cost, but the software tool allows the user the capability of overriding software estimates to provide more accurate estimates as well.

SSDSE software tool is designed to support rapid development of small satellite systems - those in the 200 kg class. This software helps designers reduce the time and effort spent on a particular design phase, however it is not developed to be all-in-one tool to create satellite designs that depend on a given mission requirements.

2.1.2 Air Force Research Laboratories (AFRL) - SatBuilder ™. The software tool named SatBuilder™ is designed to meet the needs for Responsive Space [16] in AFRL. SatBuilder™ is an artificial intelligence (AI) based toolset that provides rapid designs through wizards and interfaces with existing design tools such as Systems Improved Numerical Differencing Analyzer (SINDA), NASA Structural Analysis (Nastran), and Structural Dynamics Toolbox (SDT). SINDA is a software system for solving lumped-parameter, finite difference, and finite element (FE) representations of physical problems governed by diffusion-type equations. Nastran is a powerful general purpose FE analysis (FEA) program which is also a standard in the structural analysis field, which provides users with a wide range of modeling and analysis capabilities. SDT provides a general FE architecture and a range of specialized solvers used to study vibration problems.

OpenSat, a database and data exchange environment, is used to integrate existing tools within SatBuilder™. OpenSat also tracks requirements, related test and verification information, and PnP components. Like the approach presented in this research, the mission requirement, launch vehicle, and payload information need to be validated by SatBuilder™ in order to create a design space before the creation of the satellite bus. Overall, the SatBuilder and OpenSat provide a good foundation for future development and implementations for designing ORS satellites.

2.1.3 Star Technologies Corporation - Spacecraft Design Tool (SDT). Star Technologies developed the SDT for AFRL's Responsive Space Testbed (RST) in support of DoD's ORS initiative [19]. SDT supports all satellite design processes from mission capture to deployment by providing the user real-time spacecraft dynamics/kinematics, earth environment, and sensor/actuator models. SDT not only simulates satellite components such as sensors and actuators but also provides user the capability of adding their own subsystems (propulsion or power subsystem) to create a true software PnP environment. Figure 2.1 shows how the Responsive Space Satellite Cell in RST is configured to deliver the simulated environment to Hardware-In-Loop (HWIL) Device Under Test (DUT). As shown in Figure 2.1, SDT provides the Detailed Simulation and the Spacecraft Drag & Drop functions, as well as graphical user interface (GUI) data displays.

The SDT software tool utilizes component based simulation framework developed by Star Technologies and Microsoft's .Net Framework, which provides support for nearly 40 programming languages, on demand model loading, and rapid customization of scripts. The SDT Framework is integrated with an active Data Dictionary which provides hierarchical namespace for most objects at runtime and can be modified during the simulation. Data Items can be dragged from the Data Dictionary

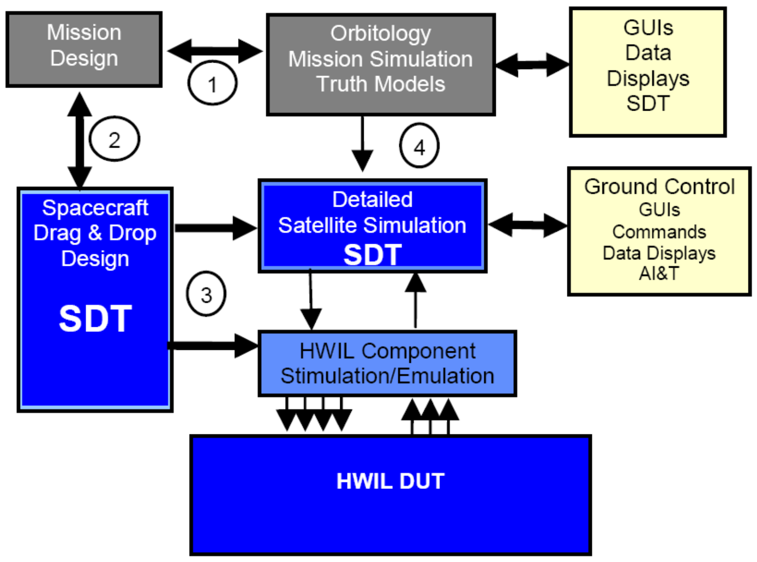

Mission Design
Orbitology Mission Simulation Truth Models
GUIs Data Displays SDT
1
2
4
Spacecraft Drag & Drop Design
SDT
Detailed Satellite Simulation SDT
Ground Control GUIs Commands Data Displays AI&T
3
HWIL Component Stimulation/Emulation
HWIL DUT

Figure 2.1: RST Functional Partitions [19]

tree view to a Monitor view to see text values, a strip chart, Mercator view, and 3D visualization. Component configuration, initialization script, visible views, and input/output (I/O) configuration information is kept in a scenario document which is loaded by a .Net application to start SDT. The Block Diagram Editor Module allows user to edit hierarchical component configurations which includes environmental, 6 DoF, visual, actuator, electrical, and propulsion building blocks.

A 3D visualization displays Earth Centered Inertial (ECI) based view of Earth with other environmental components. The camera can be assigned to any component and image views and can be tied to any location on the earth. Any number of satellites in the scenario can be displayed in multiple 3D view ports.

Overall, SDT has various features for designing and testing PnP satellites and subsystems. Since the current SDT design is based on existing satellite components, creating satellites using SDT software tool requires close user interaction with the SDT and SDT Framework iteratively. In this research, our goal is to create a "constraint based" approach for the "automatic" integration process of components to create PnP satellites that requires assuming all satellite components are prefabricated, tested, and available for integration.

2.1.4 University of Munich, Institute of Lightweight Structures (LLB) - A Software Approach for Rapid Design of Satellites. A team in the LLB developed a software approach to support and enlarge the design branch to reach a better design in shorter time [14]. The team found that it is very beneficial when the satellite design team is supported with a set of feasible or nearly feasible satellite configurations supporting basic requirements (geometric, mechanical, thermal etc). The designers also found that they could analyze the conceptional satellites and spend time focused on more specific requirements such as fastening and piping. They went on to say that comparing different designs provides a great advantage compared to traditional satellite design processes. Designers can make early decisions to reduce time for overall satellite design, thus costs can also be reduced.

Their optimization approach is composed of three basic elements: Computer Aided Design (CAD) tool (CatiaV5, Pro/Engineer), Genetic Optimization Algorithm (GAME), and Application Management System (MOSES). MOSES manages the data, administration jobs, and control execution, additionally, it provides some tools for optimization, approximation, and visualization. MOSES basically integrates different discipline specific software codes into a common environment. LLB team also uses MATLAB® to integrate CAD tools such as Catia, Pro/Engineer with FE software (Ansys, Nastran). The LLB team created an interface capable of reading and regenerating design models and it determines mass, inertia moments, component collision within models. GAME is a multiplicative evolutionary algorithm based on a Pareto-Ranking method where different objective functions are converted into to a single objective function by combining all functions into a scalar measure. A set of different solutions can be created from single solutions produced for each run.

The LLB team demonstrated the capabilities of the optimization system when they created GammaBus, which is the carrier of the Galileo System [14]. The main equipment for GammaBus consists of over 50 components of which 10 of these components are located in the payload bay. A task for configuration of those payload components is created to locate optimal positions of the components with respect to constraints such as center of gravity and possible component collision constraints. Each component could be positioned on any of the four side panels or on the top panel. A CAD model is created to represent all the geometry for the structural and equipment components. The CAD model is used in calculating the moments of inertia, center of gravity, and collision between the equipment components. In this example, it takes almost 24 hours to analyze 10,000 different payload configurations on a 20 PC cluster. Maintaining functionality and inheritability of the configuration is ensured by genetic algorithm through placing the components of the same subsystems close to each other or onto the same panel. The algorithm also considers the center of gravity, field of view, and other design constraints. After all of the geometric constraints are satisfied, stiffness and load carrying optimizations can also be done in their program.

Overall, LLB approach evaluates and creates different satellite configurations by selecting components in a reference satellite configuration through a Catia CAD tool. The LLB approach is similar to the approach discussed in this research because it provides a method to evaluate a wide variety of configurations. However, LLB approach does not allow "automatic" creation and evaluation of all possible satellite configurations based on available components in inventory; the main focus of this research effort.

2.1.5 Air Force Research Laboratory (AFRL) - Drag-and-Drop and Wizard Guided Satellite Design Approaches. In the referenced document, Lanza and others introduced a new concept for building responsive satellites [12]. In this new concept, rapid satellite integration starts with mission specifications. Tools like Analytic Graphics' Satellite Toolkit (STK) simulate the mission and predict the space environment characteristics (radiation, temperature, and solar flux) in which the satellite will operate. STK also estimates many other parameters such as RF link performance and ground site availability. Not only are the basic design specifications created for the ideal specifications, but also some constraints (mass, volume, budget, etc) that need to be taken into consideration. Capturing mission needs provides designers a great amount of insight before starting the satellite design. Designers can begin designing a satellite system satisfying the mission needs through a combination of two approaches. The first approach, “drag-and-drop”, allows designers to manually choose the components including sensors, GN&C components, structural panels, power etc, with the exclusion of items like wiring harness and software. The component may be provided from a vendor, developed by design team, or introduced later in the process.

The second part of the design process guides the designer by “wizards” from which the subsystems and eventually an entire satellite is generated. The wizard only accepts limited number of user inputs and creates a number of spacecraft configurations which can be modified later to meet the mission requirements.

In the next step, the rapid design/integration process allows the “virtual” satellite to be evaluated in a modeling and simulation program so that the virtual satellite and the original design specifications can be compared. If the differences between the mission capture and current design are not acceptable, it may be necessary to alter some aspects of the previous steps in the virtual design. Expanding the depository, increasing use of wizards or changing the parametric setting could produce better configurations.

When a satisfactory virtual design is produced, the AFRL design approach will let the user run some qualification simulations. A satellite FE model is created on which structural and thermal response predictions of the behavior in the space environment are computed. The software allows the user to change the design or specifications to create a satellite design that passes these qualification simulations. Once the design is qualified via simulation, software and wiring harnesses are automatically generated.

Lanza and others also stated that there are some challenges in the concept explained above which includes: development of a computer-aided design tool for rapid development of satellites, development of PnP depository, and minimizing the custom components used in satellites.

The concept of using a variety of software tools is good and is intended to to be part of this research effort. Overall the drag and drop wizard approach is quite similar to the approach presented in this research. However, it differs from the approach presented in this research since AFRL's approach is a manual method that requires close user interaction with software tools and focusing on creating only one satellite configuration at a time. The main purpose of the approach presented in this research is to create all possible valid "virtual" satellite configurations automatically while eliminating invalid configurations in the process based on constraints created from user inputs.

2.2 Satellite Design Approaches

So far, we have discussed specific software applications for satellite design. In this next section, we look at design approaches that may or may not be developed in code. The traditional satellite design processes typically consists of iterative steps and require significant time for integration of components software applications in an optimal way. On the other hand, in an ORS satellite design process, different methods need to be developed to significantly reduce the number of iterations and allow designers to explore the design space more quickly and efficiently. The following section describes predominantly non-software approaches for developing/designing satellites.

2.2.1 Microcosm Inc. - Space Mission Analysis and Design (SMAD) Approach. Wertz and Larson state that no single process can fully cover all contingencies in space mission analysis and design process [22]. In their book, SMAD [22], they describe a satellite design method that has evolved from years of space space exploration. In the SMAD approach, analysis and satellite design are iterative processes in that designers gradually refine both the mission requirements and methods for achieving those requirements. Table 2.1 shows a broad overview of the SMAD approach which primarily leads designers to more detailed and better defined designs in each iteration.

Table 2.1: The Space Mission Analysis and Design (SMAD) Process

Define Objective	1 Define broad objectives and constraints 2 Estimate quantitative mission mission needs and requirements
Characterize the Mission	3 Define alternative mission concepts 4 Define alternative mission architectures 5 Identify system drivers for each 6 Characterize all mission concepts and architecture
Evaluate the Mission	7 Identify critical requirements 8 Evaluate mission utility 9 Define mission concept (baseline)
Define Requirement	10 Define system requirements 11 Allocate system requirements to system elements

In Step 1, broad mission needs are defined to achieve the mission. Every mission typically has one primary objective and secondary objectives. Developers return to these broad objectives over and over in the iteration process to make sure the work done supports the space mission.

Step 2 quantifies how closely developers wish to achieve the broad objectives based on mission needs, technology, and cost constraints. Trade studies need to be performed to generate quantitative requirements. For example, the pointing accuracy of a payload to observe the Earth is related to the money spent on acquisition of that payload. There are three broad mission area objectives that can be transformed into requirements:

1. Functional Requirements which are defined by how well the system has to perform to meet its objectives,

2. Operational Requirements which are defined by how the system operates and how the users interact with the system, and
3. Constraints that limit system developers by cost, schedule, and implementation techniques.

Step 3 defines alternative mission concepts that consists of best options that are available to carry out the mission. Step 3 also includes issues such as how data will be sensed and delivered to end user, how the mission will be controlled, and the overall mission timeline.

Step 4 defines alternate combinations of mission architectures to meet the requirements of the mission concepts. All space mission architecture has following basic elements to some degree includes: Ground Elements, Mission Operations, Command Control and Communication Architecture, Launch Element, Spacecraft, Payload, Orbit, and Constellation. In a satellite system, many individual aspects influence the overall design, however, there are number of components or key parameters referred to as "drivers" that significantly influence the overall cost and performance.

Step 5 identifies the principal system drivers which influence performance, cost, risk or schedule for each alternative mission concepts. For most space missions, system drivers include number of satellites, altitude, power, component size, and weight.

Step 6 defines what a system is in detail by determining a numerical list of components for power, weight, and pointing. While characterizing the mission, system-level requirements and trades are the primary focus of the developers.

Step 7 identifies critical requirements which dominate the space mission's overall design and are responsible for determining cost and complexity of system. An example for the critical requirement for an Earth observation satellite might be location accuracy, resolution, and coverage. Developers should focus on these critical requirements to determine how good they should be made and how much money should be spent for them to achieve broad objectives. Critical requirements may differ depending on the alternative mission concepts.

Step 8 quantifies how well the design meets the critical requirements or broad objectives and provides decision makers with performance vs cost charts.

Step 9 evaluates alternative designs where one or more baseline system designs which meets most or all of the mission objectives is selected. The baseline provides a temporary milestone to compare and measure design progress. However, the baseline should not be regarded as unchangeable rather than a starting point for an iterative process. As the system design matures, the baseline becomes more firm and eventually becomes the system design.

Step 10 translates the broad objectives and constraints of the mission into specific system requirements for builders of the satellite.

Finally, in Step 11, the numerical requirements are allocated to the components. The final list of requirements that defines the quality of the job of the space mission is created by the developers.

In the SMAD approach, the subsystems are configured according to the payload. Additionally, the subsystems need to be configured in accordance with other subsystems that result from the iterative configuration approach. The SMAD approach requires multiple iterations of each step to reach to a point that all requirements are defined based on the mission type. This iterative approach consists of many integration and validation tests of components that can not be done in the short time requirements the ORS program requires.

The SMAD approach defines requirements of diverse mission scenarios. Based on mission requirements, PnP components are selected from warehouse inventory and used to create satellite configurations. Thus, one suggestion would be to use SMAD approach beforehand to identify mission requirements and design constraints to minimize the number of satellite configurations before implementing the approach presented in this research effort.

2.2.2 Arthur S. Hall - Seven Step Process. Hall's approach is comprised of seven steps: Problem Definition, Value System Design, System Synthesis, System Modeling, Optimization, Decision Making, and Implementation (see Figure 2.2). Each step of Hall's approach is influenced by the actions taken in the earlier steps. Hall's method is an iterative process, like the SMAD process, that forces refinement in each step as the process continues. The sequence allows developers to define the problem, evaluate options, and create possible solution alternatives. In the Problem Definition step, developers define the system needs and determine which elements can be altered and which elements can not be altered when solving the problem. The objectives that should be met in solving the problem are also determined in the Problem Definition step. In the Value System Design step, developers create some measure for each objective to determine how well a potential solution satisfies the objectives, moreover, the measure should support the decision makers' goals or objectives. In the System Synthesis step, some feasibility tests are applied to the alternative solutions to clarify their relevance to satisfying system needs and objectives. In order to qualify each alternative solution, Hall's approach encourages developers to create models and simulations. Once the basic modeling is accomplished in the Optimization step, different aspects of each possible solution are altered in an attempt to make each alternative as good as possible. In the Decision Making step, developers rank the alternatives based on decision maker's subjective values and decide which alternatives require further study. With alternative solutions selected, a plan of action for the next iteration through the Seven Step Process is created. Once an adequate implementation strategy is accepted by the decision maker, Hall's Seven Step Process is completed.

Overall, Hall's Seven Step Process is similar to the SMAD process in that it requires iteration and assumes component level optimization can occur. In this research effort, we want to eliminate the iterative process because it does not allow creation of responsive satellites in the time frame that ORS requires.

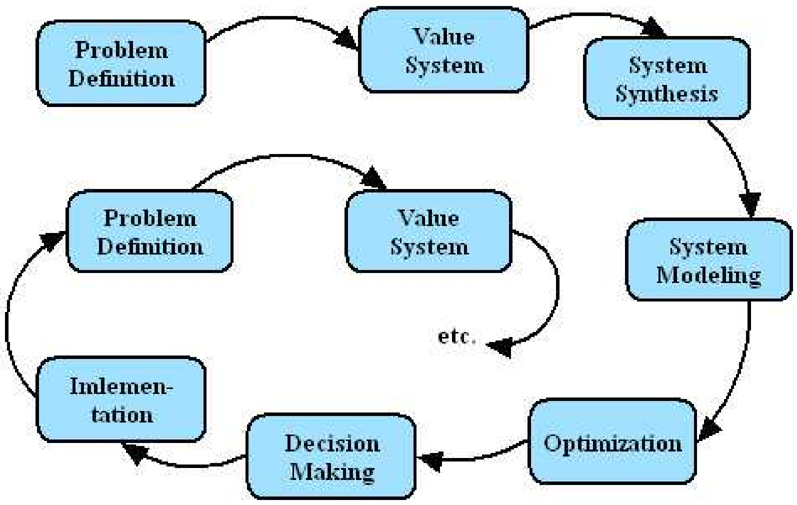


Figure 2.2: Hall's Seven Steps [7]

2.2.3 Air Force Institute of Technology (AFIT) - MODSAT Approach. From and others explain that both Hall's and the SMAD approach result in developers wasting time performing numerous iterations in the design process [7]. Additionally, the SMAD approach requires that the payload is the key focus in designing the bus. Neither Hall's or the SMAD approach provides adequate tools for designing a standardized satellite bus. An AFIT design team created a new design approach called MODSAT by customizing Hall's and SMAD approaches. The MODSAT approach is comprised of the eight steps and is described below.

The first two steps of MODSAT approach, Problem Definition and Value System Design steps, are very similar to Hall's first two steps. In the Problem Definition step, a well defined title for the problem along with a scope are created and problem constraints and needs are identified. The selection criteria for the problem solution are defined in the Value System Design step which also defines the standards by which the designers evaluate possible solutions.

The Trade Studies step evolved from the SMAD approach. Although, in the SMAD approach it is not specifically mentioned that subsystem level trades would occur in the process, MODSAT has both the system and sub-system trades occurring early in the design process. For example, whether using fuel cells or batteries along with a solar array as part of the power subsystem would require a system level trade-off study. On the other hand, the type of the power regulator used in the power subsystem is decided by both subsystem and system level trade-off studies. Any incompatibility among components found later in the design process may be hard to compensate for with simple modifications. Making trade design studies early in the design process and defining system boundaries reduces the time spent identifying the principal cost and performance drivers. Overall, they stated that designing a satellite is an art, therefore, many satellite components have to be strategically placed within the limits of a satellite structure to meet constraints such as thermal, center of mass, volume, mass, and size.

The AFIT design team uses first order estimates and relationships from the SMAD process in their Modeling Step. Time and cost savings can be best realized through analyzing models, especially when requirements or assumptions change often. Modeling usually requires three-dimensional (3D) visualization of the placement and orientation of the components. Using the base design, it is possible to create alternative designs in a timely fashion by moving, replacing, and resizing the models. System Synthesis and System Analysis steps follow the Modeling step. The System Synthesis step creates alternative solution sets and the System Analysis step score each alternative against the problem's evaluation structure. In the Decision Making step, a sensitivity analysis is performed by allowing one variable to change at a time. Sensitivity analysis helps decision makers with subjective decisions. The last step in MODSAT is the Implementation step which includes the plans and recommendations of the design team for fielding the selected alternative.

The majority of the steps come directly from Hall's seven step process with reordering of the system synthesis and modeling steps in addition to the trade studies step. The MODSAT approach doesn't include Hall's optimization step. The MODSAT approach distinguishes itself from SMAD by not focusing on requirements as the key factor for satellite bus design. Additionally, MODSAT process specifically includes a method for evaluating the values of each design alternative. Overall, the

MODSAT approach created from both SMAD and Hall's approaches allows developers a chance to create a satellite bus design without regard to a particular mission type.

2.3 Plug-And-Play (PnP) Developments

Now that we have looked at software tools and approaches related to this research, it is important to discuss recent PnP hardware developments. The PnP technology in satellite design has improved significantly in recent years. In this section, some examples of PnP technologies supporting the goals of ORS are described.

2.3.1 Air Force Research Laboratory (AFRL) - Responsive Space Advanced Technology Study (RSATS). In 2004, AFRL conducted a responsive space study to examine the benefits by having an ORS capability. AFRL's RSATS led to the identification of some key PnP technologies [15]. One of the primary areas that this study focused on is personal computers (PC)/networks. The universal serial bus (USB) is the most well-known standard that is widely used in many devices and represents a good example of the PnP concept. Once the user plugs the device into an appropriate port/socket, the USB system accomplishes identification, resource configuration, registration, and power distribution of the device for rapid use for a compatible application.

2.3.2 AFRL - Adaptive Avionics Experiment (AAE). AAE was a responsive space experiment led by AFRL, to examine the role of possible reconfigurable components to enable rapid integration [12]. Some of the research areas AAE was exploring off-the-shelf sensors that could be plugged into a receptive bus, a generic wiring manifold that can be customized/modified in seconds, and a software controlled system checkout that could reduce check-out time.

Lyke and others explained that software and wiring harnesses are the most significant barriers to PnP components. In contemporary spacecraft, even when standard interfaces are used, it is often necessary to customize both the hardware and software for each spacecraft component. Each combination of these standard elements requires custom wiring harnesses in addition to custom software that could require many months to create.

Today, it is a very common practice to allocate several months in a development schedule to integrate components produced in many different locations by different people/companies [12]. Adaptive avionics allows complex configurations through combining "off-the-shelf" state reconfigurable PnP components and subsystems. Machine-negotiated interfaces, one that allows self-configuration and self-organization, permits developers to create more reliable systems by reducing error-prone human interpretation.

The idea of the "drag-and-drop" satellite was introduced by AFRL, as explained in the Section 2.1.5, and it requires that most satellite components be pre-built and reconfigurable so that they could easily be programmed to perform different tasks. The components are engineered to be rapidly assembled by using structural standards and reconfigurability concepts such as adaptive manifolds, and self-aligning connectors. These are some of the key concepts on which this research effort is built.

2.3.3 AFRL - Adaptive Wiring Manifold (AWM).

AWM can be used as a standard prefabricated wiring harness which can be configured rapidly before launch [12]. The AWM provides anomaly mitigation during both integration phase of the satellite and on orbit because it is comprised of hundreds of relays which are accessible through software commands. Even though this is an interesting technology, it is not considered in this research.

2.3.4 AFRL - Appliqué Sensors (AS).

AS provides a mechanism for sensors and actuators that can be networked with a common interface through AWM so that their placement on the spacecraft is not restricted to specific locations [12]. AS has distributed intelligence providing self-configuration and self-healing components. In

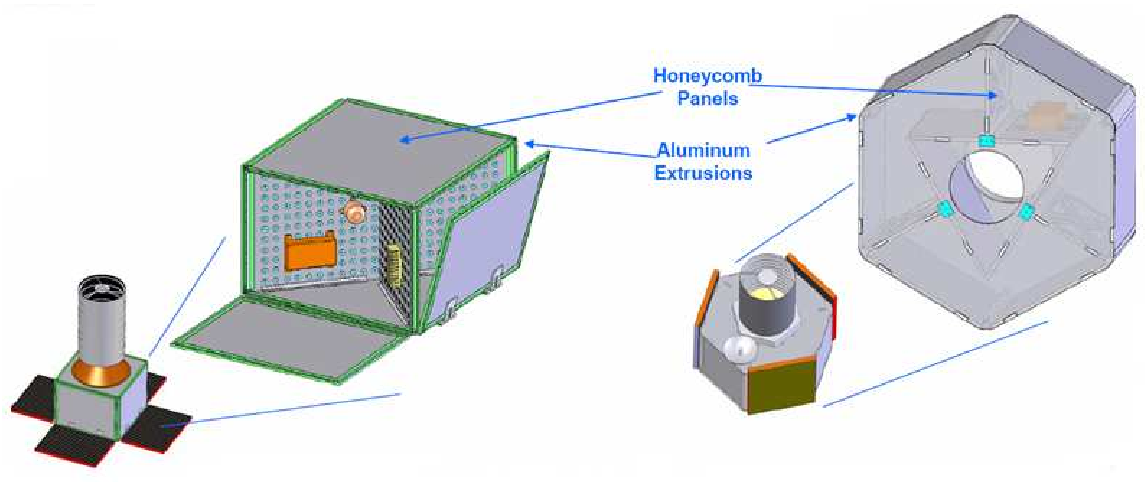


Figure 2.3: CSA's Structural Architecture - Demonstrates how a standard set of component can be assembled to create varying satellite structure [1]

the AAE experiment, it is proposed to test simple dose rate and total dose radiation sensors along with some other sensors for monitoring health and environment conditions such as power, vibration, and temperature.

2.3.5 CSA Engineering, Inc. - Robust, Reconfigurable Structural Architecture.

CSA Engineering, Inc. developed a structural architecture that provides developers the capability of assembling different configurations based on the needs of mission, launch vehicle, and payload using a set of structural components [1]. The approach described as a "2x4 and plywood" approach which uses prefabricated aluminium/honeycomb panels and sets of frame elements. The frame elements are used for keeping panels attached to each other. Additionally, the same panels can be used in different configurations. As shown in Figure 2.3, similar panels are used to create rectangular and hexagonal busses while frame elements and bottom and top panels changes between the two configurations. Since prefabricated structural components can be assembled in a very methodical and controlled manner, the architecture provides extremely tight alignment tolerance, thus an easier assembly process.

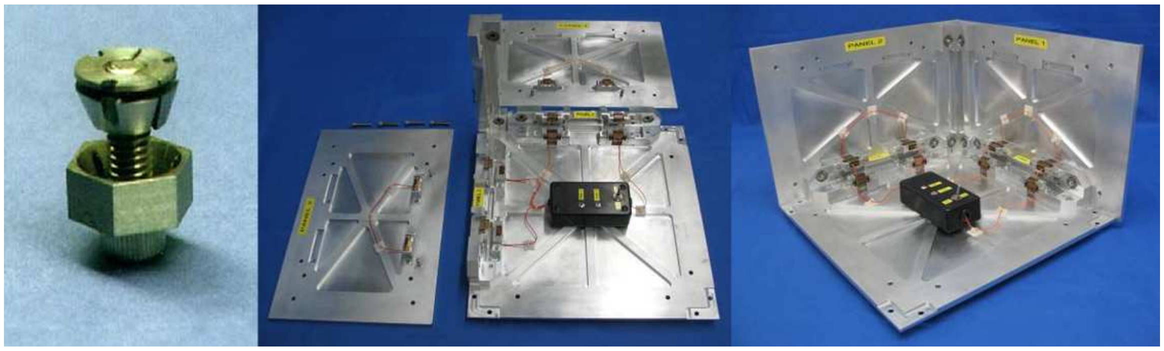


Figure 2.4: PnPSat Structure in Flat-Sat Configuration [1]

2.3.6 Honeybee Robotics Spacecraft Mechanism Corp. - Attachment Mechanism for Mechanical/Electrical/Thermal Connections. Honeybee focused on simultaneous mechanical and electrical connections between structural components to assemble a complete satellite within a matter of hours [1]. Honeybees's design is based on the Quick-Insertion-Nut (QIN), which allows a bolt to be inserted into a mechanism acting as a nut. After the satellite passes initial check-out, the structure is locked down by simply rotating each bolt less then one revolution with a torque wrench. Figure 2.4 shows pictures of the QIN and a demonstration model of the rapid attachment of a "box" satellite.

When the QIN is torqued to the designed value, an elastic, as opposed to plastic, deformation occurs in the nut. Therefore, the cycle is repeatable up to 1000 cycles without any bolt back out. The QIN also behaves extremely non-linearly when not fully tightened, which is an important feature supporting the structural health monitoring (SHM) efforts for rapid check-out of responsive satellites. If there is a fault, SHM provides a capability of pinpointing the location, type, and severity of the fault [2]. Detecting rapidly the location of the fault (which may be as simple as tightening a bolt) is very helpful for immediate repair of satellite.

2.3.7 AFRL - Space PnP Avionics (SPA). AFRL, in collaboration with other government, industry, and academic organizations started the Responsive Space (RS) program to research how it would be possible to create space systems in a shorter time [8]. The main idea is to accelerate component integration by employing intelligent interfaces referred to as Space PnP Avionics (SPA) in the design process. SPA components are similar to USB components of today's PCs in the terms of self description and self-organization. The xTEDS mechanism (XML-based electronic datasheets) in SPA components makes it possible for components to carry their own documentation. In principal, SPA may eliminate artificial constraints caused by a fixed network structure by using self-forming networks. The Satellite Design Automation (SDA) framework helps user to translate the needs of the developers into a buildable satellite system which meets that need. The SDA framework guides the designer to determine mission requirements and constraints as well as orbit and flight characteristics of the mission which are developed based on available launch options. A set of tools helps developers to find a "good enough" spacecraft configuration based on SPA components supporting mission, orbit, and flight characteristics.

The study concluded that the technology and standards to be developed in a multi "spiraled" approach [15]. Four development phases were created starting with Generation "zero" (Gen 0) which studied the development of near-term hardware interface concepts based on the commercial off-the-shelf (COTS) technologies. The authors stated that Gen 0 implementations are not suitable for the harshest radiation environment, but they provide a good insight for early PnP component development. The Generation 1 (Gen 1) study had the same focus as Gen 0, with addition of radiation-hardened components. The Generation 2 (Gen 2) study focused on moving from centralized architecture to a distributed architecture while maintaining the backwards compatibility with Gen 1. The Generation 3 (Gen 3) study focused on implementation of "geographic awareness" of components, adaptive wiring manifold, and self-awareness.

The authors stated that creating a responsive spacecraft rapidly is difficult because of the technical challenges among the subsystems, development of the avionics, and software [15]. Interfaces among components, wiring harness, and auto-generated codes are the key elements for bringing together the components of payload, spacecraft and/or launch vehicle. In the personal computer industry, there is significant standardization effort in PnP components. PnP components in space industry should result in components becoming low-cost, rapidly integrable, and more reliable in time. Reconfigurability plays an important role in reducing the time necessary to integrate PnP components.

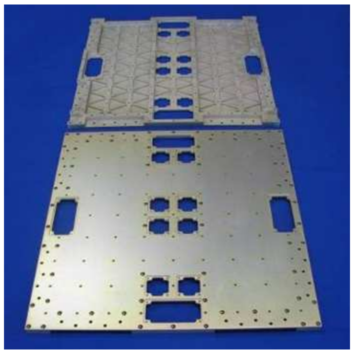
The SPA approach fully supports creating complex configurations of virtually any sensor or actuator type. This behavior makes the network easily reconfigurable and robust. SPA is defined as an interface-driven standard supporting the rapid development of spacecraft busses and payloads. The SPA standards use commercial standards (such as USB) with additional specifications engineered to accommodate special constraints such as high power, synchronization, and fault tolerance. Interconnect standards including USB, Spacewire, and Ethernet are not themselves sufficient enough to achieve PnP. SPA-U, based on the existing USB (version 1.1) interface standards, not only supports 12 Mbps data transport but provides power up to 3A at 28 V and synchronization using 1 Hz sync pulse via supplemented conductors. Spacewire based SPA (SPA-S) is an European Space Agency (ESA) standard very similar to SPA-U device with higher speed data transport and power handling. SPA-S components are more complex, but smaller than SPA-U components.

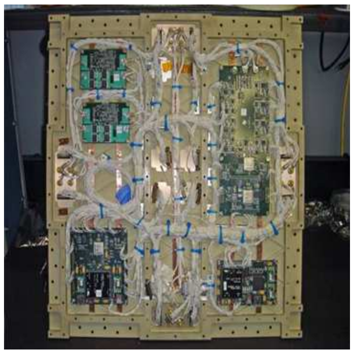
Presently, many people designing payloads suitable for PnP satellites bring along concerns of additional cost, complexity, and risk. It is unlikely that two independently developed components will work efficiently in the same network without standardized communication interfaces. To break conventional satellite manufacturing legacy, the AFRL team started an experimental PnP satellite (PnPSat) design and has stated that they have encountered many obstacles; thus they have concluded a “clean sheet” approach is more suitable for constructing a responsive satellites. They started by applying PnP principles to mechanical, electrical, and software interfaces in the satellite.

The modularity and complexity-hiding are the two most important principles demonstrated in PnPSat. The current PnPSat design has 48 standard PnP mechanical and electrical interfaces located on either interior and exterior surfaces providing flexibility to 25 PnP components [8]. The research effort presented in this thesis is based around SPA standards that are fully utilized in AFRL's PnPSat. PnPSat represents the future of ORS satellites because it enforces modularity of components.

2.3.8 SpaceWorks, Inc. - PnP Satellite (PnPSat) Structure and Multifunctional Panels. The goal of SpaceWorks's PnPSat is to develop a highly capable satellite bus that can accept large number of space experiment via standard electrical and mechanical interfaces [1]. The main focus of this approach is the development of clamshell panels where SPA electronics can be housed internally. The exterior of the panels provides a set of standard electrical and mechanical interfaces for rapid satellite component integration. Figure 2.5(a) shows the aluminum panels with an iso-grid internal structure. The empty space within the panel accommodates electrical components and harnessing as shown in Figure 2.5(b).

SpaceWorks and AFRL focused on the standards for mechanical and electrical interfaces to enhance component development. The interior and exterior surfaces of spacecraft panels have 5 cm x 5 cm grid pattern for eight fasteners for mechanical mounting of all components and experiments. Each of the six spacecraft panels has eight electrical endpoints that serve as the electrical interface to the PnP network. Figure 2.5(c) shows three fully integrated structural panels with two reaction wheels assemblies and some other components (torque rods, sun sensors, magnetometer) mounted on for Attitude Determination and Control Subsystem (ADCS) testing. The integration of components is easily conducted by simply mounting the components into place and connecting electrically via standard electrical connectors. The panels similarly connect to each other creating a structure that is able to collaborate with satellite components and subsystems.


(a)


(b)

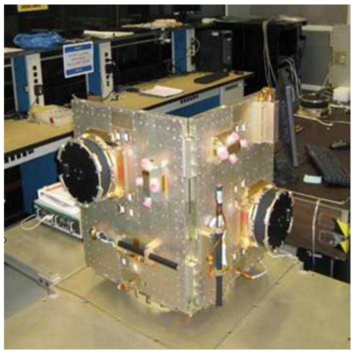

(c)

Figure 2.5: Pictures of SpaceWorks' panels
(a) external and internal structure of the clam-shell pair,
(b) internal electrical wiring and circuit boards, and
(c) standard electric/mechanic interfaces

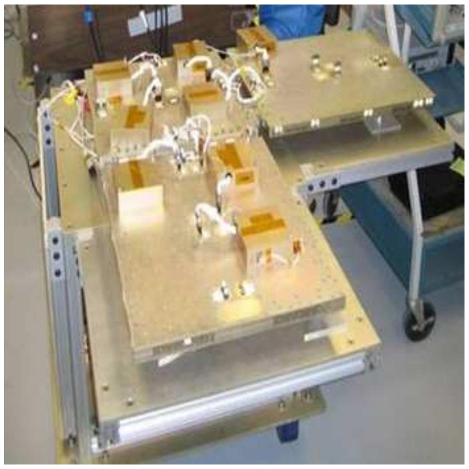


Figure 2.6: PnPSat Structure in Flat-Sat Configuration

If all components are available and attachment points are known, the spacecraft structure can be fully assembled in less than one hour as a result of "flat-sat" design approach demonstrated in Figure 2.6. Hinges located on five of the panel-to-panel joints allows any individual panel to be opened in less then 10 minutes permitting access to the satellite interior. The research presented in this document is based on this PnPSat design, where the approach presented is designed to automatically determine the locations of every component so the users can rapidly determine the best configuration that meets their needs.

III. A Constraint Based Approach for Building Operationally Responsive Satellites

This chapter presents the key methodologies used in this research effort. Section 3.1 explains some of the key features of object-oriented programming (OOP) directly related to this approach. Section 3.2 provides a brief explanation of the Gaussian probability distribution function (PDF) that is used to generate a component's property values such as mass, cost, etc. The principles of multiplication, permutations, and combinations that apply to this component configuration problem is introduced in Section 3.3. An overview of the approach created for this research effort is given in Section 3.4. Some key elements of the constraint based approach presented in this research like Virtual Warehouse, 3D Environment, and Graphical User Interfaces are discussed in Sections 3.5, 3.6, and 3.7, respectively. Finally, Section 3.8 of this chapter covers the key steps of the approach to generate valid PnP satellite configurations from known components.

3.1 Object Oriented Design

The purpose of object-oriented design (OOD) is to reduce large problems into smaller and more manageable problems by creating classes with a simple and usable interface while not exposing the users to the internal operations of the class [20]. Both *Reusability* and *Extendability* are the two major features of OOD that are used extensively in this research. *Reusability* can be defined as the likelihood that a segment of source code can be used again to add new functionalities with slight or no modification. *Extendability* is the ease with which the software that can be modified to increase functional capacity [3]. Using OOD helps programmers design software with extension points so that new programmers can add new functionality to existing software with a minimal of effort and reusing the previous codes as much as possible.

A simple analogy to explain classes and their contents is explained next. Suppose we want to drive an automobile and make it go faster by pressing down on its accelerator pedal [5]. The automobile is an extremely complex mechanical device through which the user interfaces with it by applying inputs into the accelerator pedal, brake pedal, and steering wheel. The pedal “hides” the complex mechanisms that actually make the automobile go faster, just as the brake pedal “hides” the mechanisms that slow the automobile, and the steering wheel “hides” the mechanisms that turn the automobile. By looking at the blueprints, we can see how the accelerator pedal relates to all other components in the automobile.

In Java programming, classes (blueprints of accelerator pedal, brake etc.) are designed to perform the classes’ specific tasks that are called methods (go faster, slow down, turn right/left). If you want to perform a task, a method must be created. The method describes the mechanisms that actually performs the tasks. The method also hides the complex tasks that it performs from its user, just as the accelerator pedal of an automobile hides from the driver the complex mechanisms of making the car go faster. Just as we cannot drive a blueprint of an automobile, we cannot “drive” a class. We must first build an object of a class (actual automobile) before we can get a program to perform the methods.

One of the primary features of object-oriented programming (OOP) is *inheritance*. *Inheritance* provides software reuse in which a new class is created by absorbing an existing class members and enhancing them with new or modified capabilities such as altering inherited method or adding new attributes. Attributes consist of a name and value to define the properties of an object. When creating a class, rather than declaring completely new members, the programmer can decide that the new class should inherit the members of an existing class. The existing class is called the superclass, and the new class is the subclass. Each subclass can become the superclass for future subclasses.

A subclass normally adds its own additional fields and methods. Therefore, a subclass is more specific than its superclass and represents a more specialized group of objects. Typically, the subclass exhibits the behaviors of its superclass and additional behaviors that are specific to the subclass. For a full introduction and discussion of OOP refer to references [5] and [17].

3.2 Gaussian Probability Distribution Function (PDF)

Currently, PnP hardware standards have not been fully established in the responsive space (RS) industry to the point that a repository of PnP components exists. Also, many of the properties of PnP components will be proprietary and may not be available. Therefore, a virtual repository is created for this research using Gaussian PDFs to create normally distributed sets of PnP components and their attributes based on actual components. We are assuming that in an actual software implementation of this approach, the contractors who create the PnP component, will provide the key properties such as mass, reliability, mass moments of inertia, etc.

The Gaussian PDF, also called "normal distribution" and described as a "bell-shaped curve", is one of the most commonly used distributions. The Central Limit Theorem states that under a wide range of circumstances the PDF that describes the sum of random variables tends towards a Gaussian distribution as the number of terms in the sum approaches infinity [9]. A random variable x is said to be normally distributed with mean μ, variance σ^2, and standard deviation σ if its PDF is

$$\varphi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}. \tag{3.1}$$

Figures 3.1 and 3.2 show a variety of Gaussian PDF distributions.

Nearly 68% of values drawn from a normal distribution are within one standard deviation; about 95% of the values are within two standard deviations; and about 99.7% lie within three standard deviations, as shown in Figure 3.2. In this research, all values such as mass, reliability, and quantity are randomly generated for every component using Gaussian PDFs that have means and standard deviations estimated from actual small satellite component data [23]. The Java Math Library provides a pseudorandom Gaussian distribution function where the mean is 0.0 and the standard

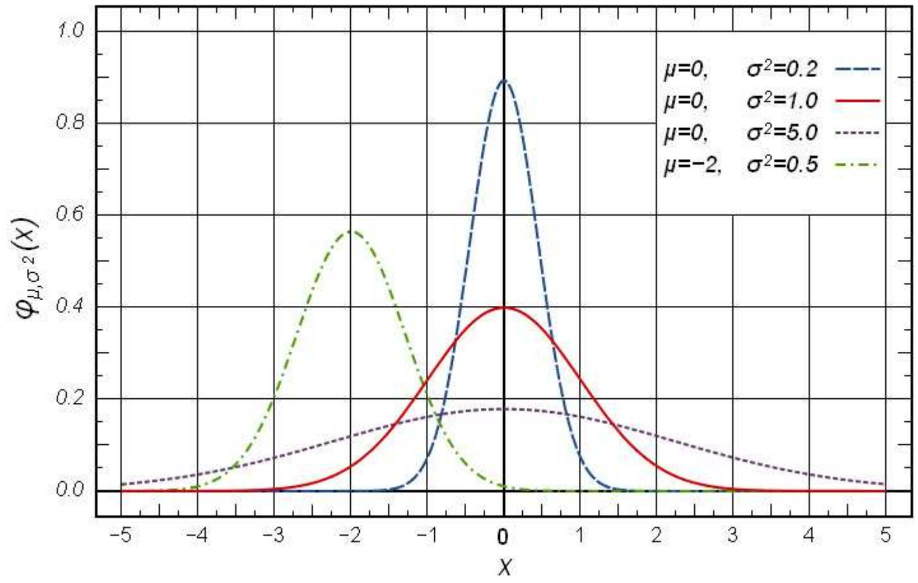


Figure 3.1: A selection of Normal Distribution Probability Density Functions (PDFs) [4]

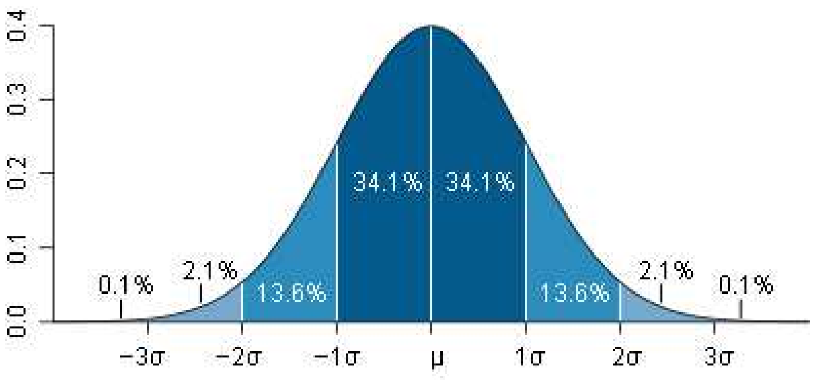


Figure 3.2: Gaussian PDF [13]

deviation is 1.0. This library is modified to generate Gaussian normal distribution numbers based on different means and standard deviations, see Appendix A Listings A.1 and A.2.

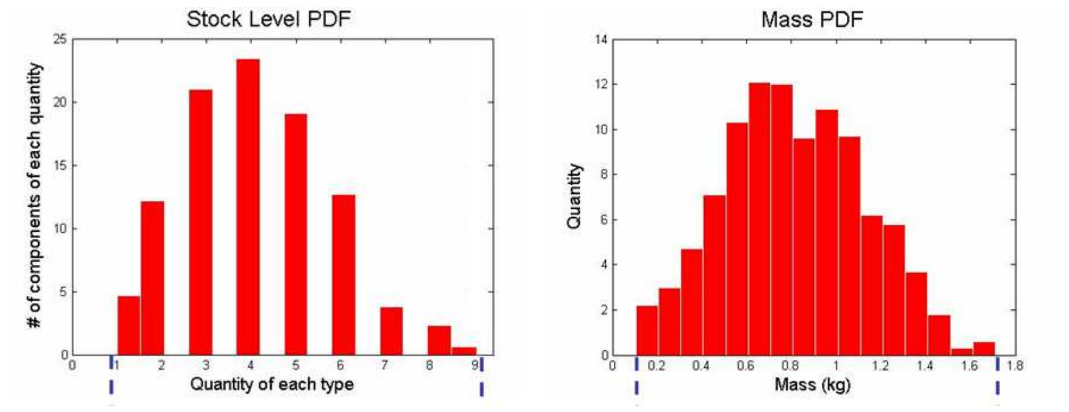
Gaussian distributed x values, which are used in creating a virtual repository, are generated in this research based on given set of actual component data by following calculation

$$x = \mu + G(\sigma/3) \tag{3.2}$$

where G is a Gaussian distributed value with mean of 0.0 and standard deviation of 1.0. Dividing the standard deviation σ derived from the actual component data by 3 generates numbers that lie within calculated standard deviation with 99.7% probability.

For the purposes of showing an example of how the Gaussian PDF distribution is used in this approach, a library consisting of 100 components is created as shown in Figure 3.3. One of the attributes of the components in the virtual repository is the *Availability* that defines the quantity of a particular component. As shown in Figure 3.3(a), the quantity of each component varies between 1 and 9 and follows a Gaussian distribution. Since the Gaussian PDF can produce any real valued number, randomly created numbers outside minimum and maximum values are filtered. The boundaries applied to *Mass* attribute is shown with “vertical dashed lines” in Figures 3.3(a) and 3.3(b).

To create random values for the *Availability* attribute, an exponential distribution might be better suited and could have been easily implemented. However, the purpose of generating random values here is to create a virtual repository that assigns realistic attributes to each component in the library and the main purpose of this research is focused on developing a new approach for designing PnP satellites.



(a) (b)

Figure 3.3: Component Values Generated using Gaussian PDF

3.3 Multiplication Principle, Combinations, and Permutations

The approach developed in this research focuses on creating satellite configurations by combining component combinations which are created from “like” (same model and manufacturer) components and component location combinations which are created based on component combinations. The “multiplication principle” and “combinations”, which are described below, are used to create component combinations and component location combinations, respectively.

The “multiplication principle” can be best explained with a simple example. Assume we have a task that consists of r operations. Operation 1 can be performed in m_1 ways, operation 2 can be performed in m_2 ways, operation 3 in m_3 ways, and so forth. Then, the task can be performed in

$$N = m_1 \times m_2 \times m_3 \times m_r \tag{3.3}$$

ways.

Similarly, suppose that we have one type of sun sensor, three types of earth sensors, and four types of thrusters with unlimited quantities to select for the Attitude Determination and Control Subsystem (ADCS). Assuming the user wants to have

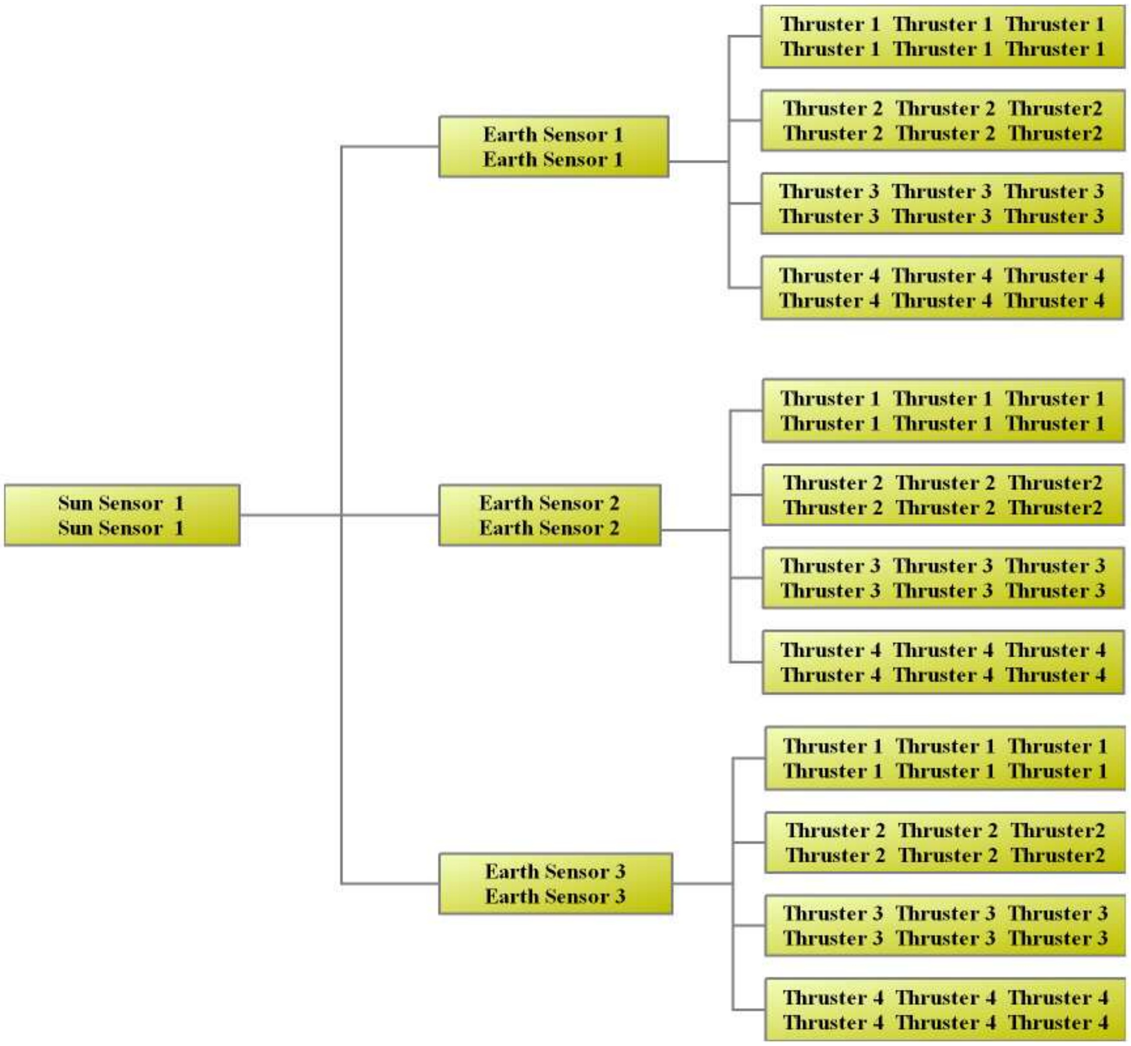

Sun Sensor 1
Sun Sensor 1
Earth Sensor 1
Earth Sensor 1
Earth Sensor 2
Earth Sensor 2
Earth Sensor 3
Earth Sensor 3
Thruster 1 Thruster 1 Thruster 1
Thruster 1 Thruster 1 Thruster 1
Thruster 2 Thruster 2 Thruster2
Thruster 2 Thruster 2 Thruster2
Thruster 3 Thruster 3 Thruster 3
Thruster 3 Thruster 3 Thruster 3
Thruster 4 Thruster 4 Thruster 4
Thruster 4 Thruster 4 Thruster 4

Figure 3.4: Example of Multiplication Principle

two sun sensors, two earth sensors, and six thrusters integrated into the satellite configuration, a total of twelve different combinations can be created as shown in Figure 3.4.

On the other hand, creating component location combinations is a "combination" problem for two reasons. First, we can not assign more than one component to the same location. Second, the order of component locations does not matter. Because of creating component combinations from "like" components, switching the locations of "like" components will not create a new configuration. For example, assume we want to create satellite combinations from a component configuration consists of three

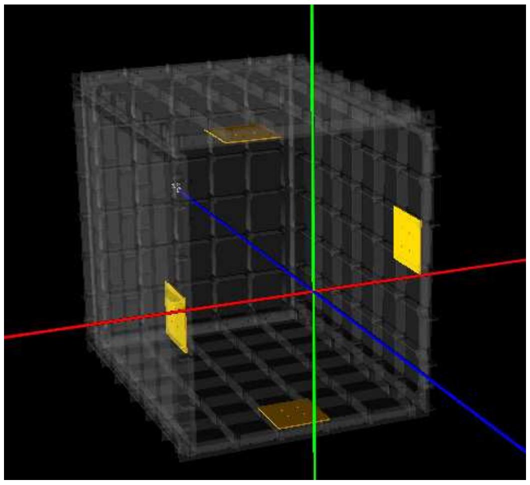


Figure 3.5: Four Color Coded Component Locations

"like" components labeled C C C, and a component location combination consisting of three component locations, L1 L2 L3. The satellite is composed of matching components with component location that a sample match would be C-L1 C-L2 C-L3. This configuration will not change, even though we switch order of the component location to L2 L1 L3.

"Combinations" can be defined as an arrangement of r objects, without regard to "order" and without "repetition", selected from n distinct objects and is denoted by

$$ {}_nC_r = \frac{n!}{(n-r)!r!} \tag{3.4} $$

where a combination of n objects is taken r at a time.

For example, assume that we want to create component location sets for sun sensors. Suppose that we want to use two sun sensors in the satellite and on the selected satellite structure we have four locations that sun sensors can be connected, as shown in Figure 3.5. From Equation (3.4), ${}_4C_2$ is six, meaning that six different combinations of two sun sensors in four locations can be created as shown in Figure 3.6.

Components and locations of components on the satellite are the key items that must be determined when building PnP satellites. In this research, to create combi-

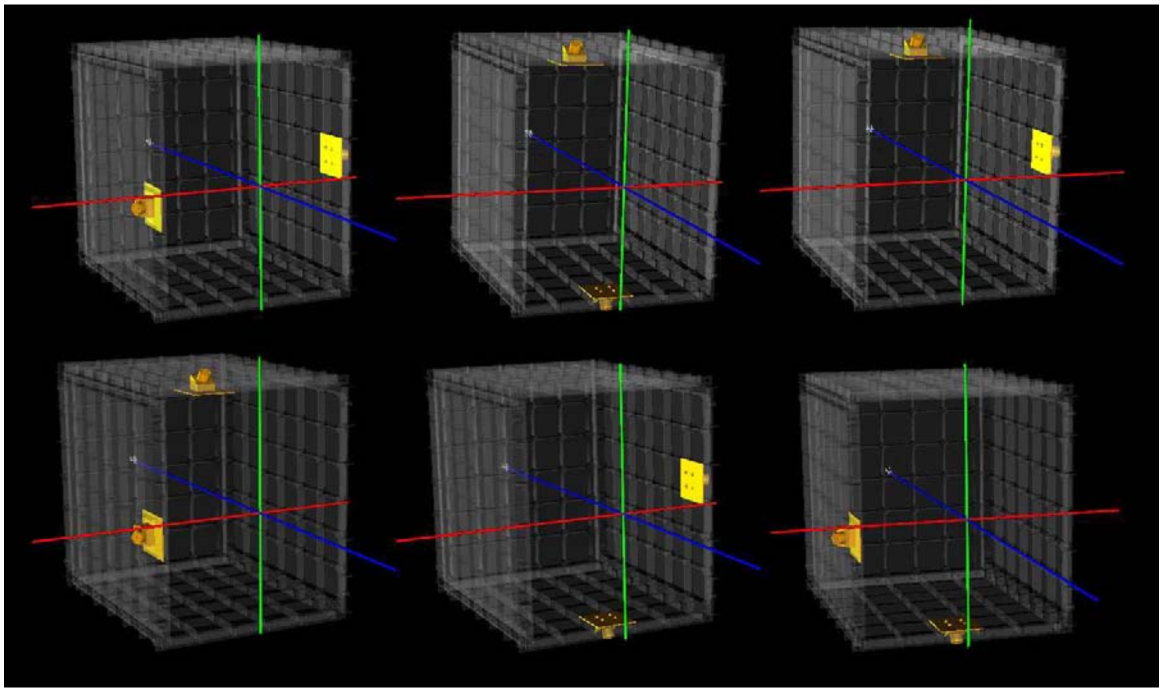


Figure 3.6: Example of Six Different Combinations of two Sun Sensors in Four Locations

nations of components and their locations we first selected the components and then determined component locations with different “orders” for those selected components. Locations with different “order” creates different satellite configurations from same components. Another approach could be created by components with different “orders” and then assign those components to selected locations. However, the second approach is not recommended because there will be thousands of components in the warehouse, and creating component combinations regarding the “order” of components creates extremely large numbers of combinations to be evaluated.

In our approach, if we allow use of “various” (different model or/and manufacturer, but still performs same function) components in the configurations and “order” of components does not matter, the number of component combinations can be calculated using following equation

$$_nC_r = \frac{(n+r-1)!}{r!(n-1)!} \tag{3.5}$$

where a combination of n objects is taken r at a time allowing repetition of components.

A simple calculation can be made to calculate the total number of configurations in the cases where we allow "various" components. If we assume that the order of the components in the component combinations does not matter and allow using "various" components, finding the number of component combinations become a "permutation" problem that can be calculated by

$$P = n^r \tag{3.6}$$

where n is the number of components to choose from and we choose r of them. Because different component combinations that consist of same components in different orders, "order" doesn't matter in creating location combinations. In a situation where permutations are used to calculate component configurations, finding the number of location configurations becomes a "combination" problem that can be calculated by Equation (3.4). For example, assume that we want to create component combinations consisting of two components from C1 C2 C3 and component locations for each combinations from L1 L2 L3. When we allow using "various" components and regarding the orders of the components, the list C1 C1, C1 C2, C2 C1, C1 C3, C3 C1, C2 C2, C2 C3, C3 C2, C3 C3, represents the nine component configurations, as calculated from Equation (3.6), and for each configuration L1 L2, L1 L3, L2 L3, three component locations, Equation (3.4), can be created. The multiplication of component permutations and location combinations gives us the total number of potential configurations.

3.4 Overview of the Constraint Based Approach

From the previous Section 3.3, it is clear that "state explosion" -a condition of extraordinarily large numbers of configurations typically caused by wide variety of components and locations- is a problem. To over come this problem, this research establishes a “constraint based” approach for creating manageable number of satellite configurations from numerous components and component locations. “Constraint based” refers to the idea of filtering components, component locations, or combinations based on constraints created in different levels of the configuration process of building a PnP satellite. Some key techniques and methods of this approach have been implemented in software to test the approach, see Appendix A. While discussing the methods, some sample implementation screen shots of the approach are included to demonstrate what a commercial implementation of this approach might look like. In the approach created from this research, properties of all PnP components and satellite structures that houses PnP components are efficiently stored in a database referred to as the ObjectDB [18].

Various constraints are derived from the user inputs at different levels of the configuration process as shown in Figure 3.7. User inputs might be very general information such as mission type, mission duration, or very specific such as the selection of a particular component and its respective location on the satellite that is determined beforehand to be used in all design configurations. The more inputs a user provides related to a PnP satellite, the more constraints can be created and it might be possible to reduce the number of possible valid PnP satellite configurations to manageable numbers or even to zero (indicating no feasible solutions exists). In this research, the user provides constraints such as mass, cost, reliability, number of each components, mission type, mission duration, etc. Available constraints are applied whenever it is applicable to minimize the number of satellite configurations. It is understood that more user inputs would be required in a commercial implementation of this approach.

The PnP satellite structures manufactured by AFRL and SpaceWorks Inc. have many connection locations that PnP components can be attached to, see Section 2.3.8. Satellite configurations are created by assigning components to the locations on the satellite structure as shown in Figure 3.8.

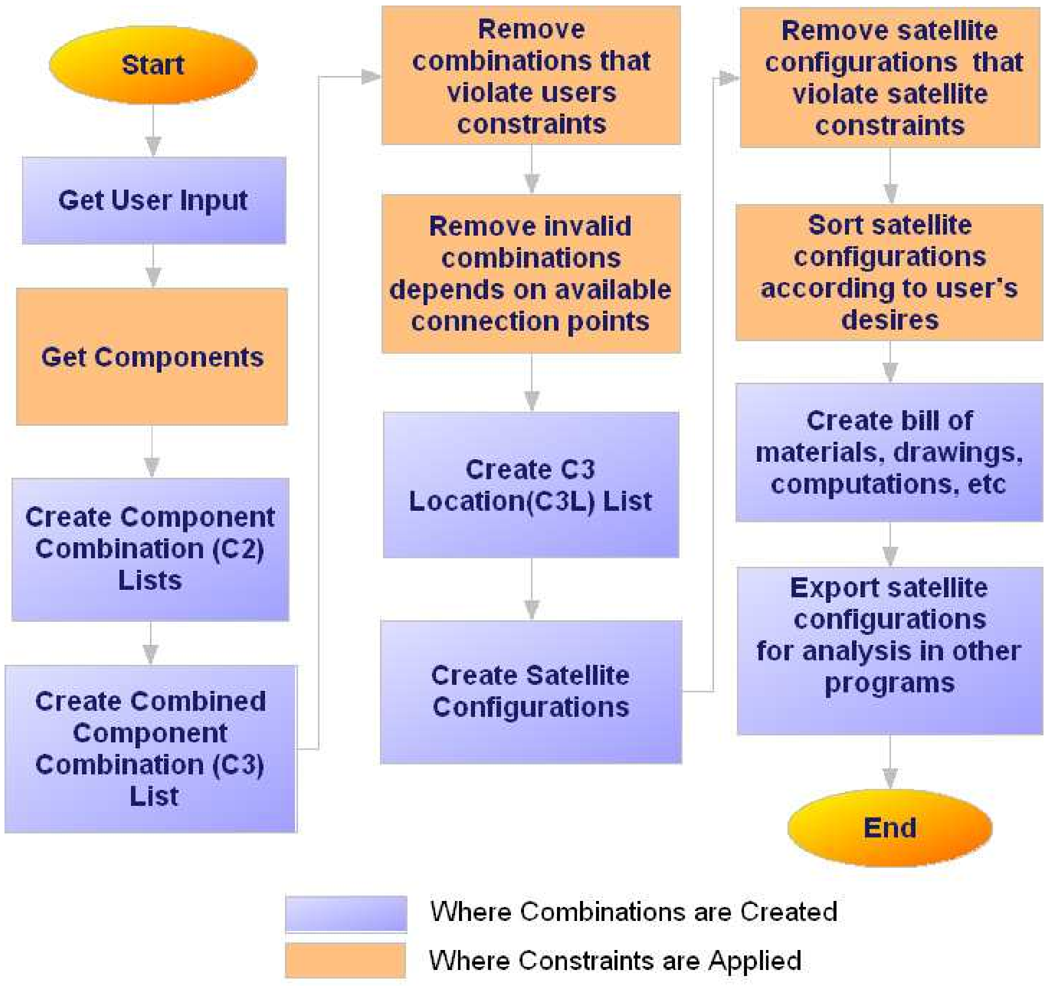

Start
Get User Input
Get Components
Create Component Combination (C2) Lists
Create Combined Component Combination (C3) List
Remove combinations that violate users constraints
Remove invalid combinations depends on available connection points
Create C3 Location(C3L) List
Create Satellite Configurations
Remove satellite configurations that violate satellite constraints
Sort satellite configurations according to user's desires
Create bill of materials, drawings, computations, etc
Export satellite configurations for analysis in other programs
End
Where Combinations are Created
Where Constraints are Applied

Figure 3.7: Overview of Constraint Based Approach

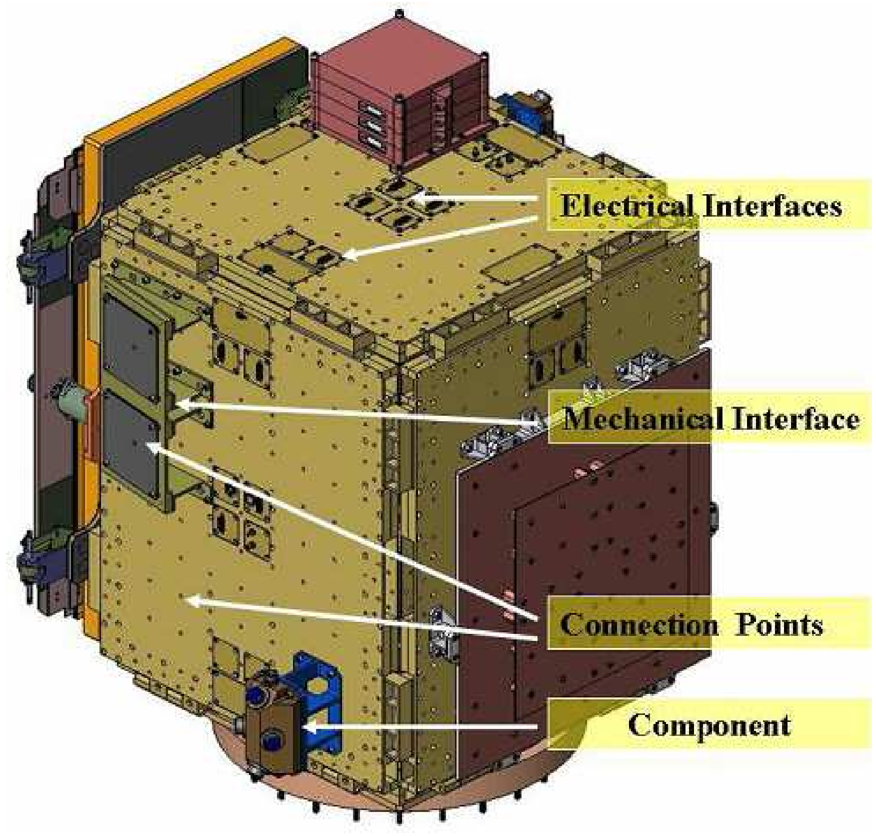


Figure 3.8: Two Panels of PnP Satellite with various Components [1]

Each satellite configuration is saved to a database and the various configurations are sorted based on cost, weight, and/or other variables. It is very likely that some components with similar properties and different part numbers exist in the warehouse. Different satellite configurations created from those similar components will have very similar properties such that those satellites can be assumed virtually the same with respect to cost, mass, etc. In this approach, configurations can also be eliminated when properties are very close to each other. In the end, the goal is to rapidly find satellite configurations that meet the user's needs from which the user can compare and contrast to determine the best configuration.

3.5 Virtual Repository - PnP Component Library

PnP components will be created by a variety of contractors/companies using different manufacturing methods that will adhere to common standards developed by AFRL (Section 2.3) to ensure compatibility. We assumed that all components are prefabricated, tested, and validated beforehand, so that users don't spend time validating every component/subsytem when they are needed. The time spent testing and validating should be done beforehand to meet the goals of the 6-day satellite build [11]. The virtual repository contains every components' computer aided design (CAD) drawings and attributes such as mass, center of mass (CM), moments of inertia (MOI), reliability prediction, etc. Additional component attributes may be included and stored in a commercial implementation of this approach.

A data structure composed of different level of entities is created to accommodate all elements of the virtual repository. Each entity level accommodates a group of components having the same attributes and functionalities. Any component used in composing a satellite belongs to a minimum of one entity level in the designed data structure. In this document, names related to data structure such as entities and attributes are italicized to prevent any confusion with actual meanings. For example, the italicized word *Component* represents the entity in the data structure while "Component" refers to the physical part. *Component* is the highest level entity from which all other lower level entities inherit. Lower level *Components* are categorized based on their characteristics and functionalities to form some other entities such as *Sun Sensor*, *Earth Sensor*, *Thruster*, or *Reaction Wheel.* Similarly, entities in data structure like *Sun Sensor*, *Earth Sensor*, and *Thruster* represent actual component types such as sun sensor, earth sensor, and thruster in the warehouse. All entities inherit most common attributes such as *Mass*, *Cost*, *Availability*, *Reliability*, etc. from the *Component* entity. Currently, the *Component* attributes described in Table 3.1 have been created to test the proposed approach and more attributes can be added to this initial list as new ones are identified.

Table 3.1: *Component* Attributes

***Component* Attribute**	**Description**
Id	A unique name for the *Component*
Availability	Quantity of like components (same *Id*) in the PnP component library
Connection Type	Common interface that provides physical bonds, data transfer connection to the bus, electric power through *Connection Points.* In order to attach a PnP component to a connection point, the *Connection Type* attributes of *Components* and *Connection Point* of the satelite structure must match
Cost	Purchase or replacement cost of a component in the PnP component library, it doesn't include the labor cost of integration
Dimensions	Length, width, and height of the component in meters
Mass	Mass of the component in kilograms
Orientation	The orientation that the component can be rotated without any restriction in functionality. Orientation defines pointing directions with respect to satellite attitude such as +X, -X, +Y, -Y, +Z, -Z, where +Z is assumed the zenith and -Z is the nadir direction. In order to attach a component into a connection point, the *Orientation* attribute of *Connection Point* of the satellite structure should support the *Orientation* attribute of *Component*
Reliability	Reliability of the component based on the failure rate provided by the manufacturer. Overall reliability of a satellite system can be estimated based on position (parallel/serial) of components and subsystems in the architecture of satellite system

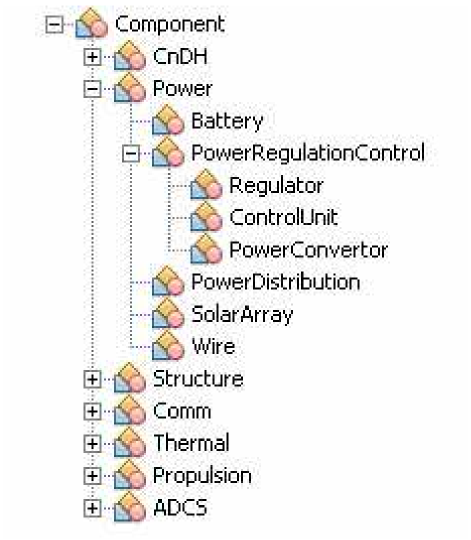


Figure 3.9: Hierarchy of Entities

All entities inherit common attributes from *Component* and have some other attributes specific to their characteristics. For example, a sensor component, which is a member of the *ADCS* entity, has the *Accuracy* attribute where a component that belongs to *Power* doesn't have. Entities like *Sun Sensor*, *Earth Sensor*, and *Thruster* can form another sub-entity if they can be grouped based on more specific functionality. As shown in the Figure 3.9, *Control Unit* is a *Component* and belongs to both the *Power Regulation Control* entity and *Power* entity at the same time. The *Control Unit* can also be grouped into other entities. Lower level entities inherit all attributes from higher level entities including all *Component* attributes. Entities like *Power*, *ADCS*, *Propulsion*, etc. represent subsystems and have common attributes such as *Center of Mass* (CM) and *Inertia Tensor* (I). Estimations of CM and I can be stored and used when it is needed. By combining OOD and the satellite subsystem hierarchy, we created a data structure that accommodates a wide variety of different properties of PnP components. Having different attributes at different levels in the system makes coding and calculations easier because it hides the complexity of data structure at different levels.

PnP Components and Satellite Structures that accommodate those components are the two types of physical elements from which a PnP satellite is created. We are assuming that the payload is also PnP compatible and is treated as a component. A Satellite Structure is comprised of panels with various connection points that accommodate electrical and structural interfaces for PnP components to be connected. In Figure 3.8, actual connection points and different components are shown for two panels of a PnP satellite. We assume that there will be different sizes of the Satellite Structure available for different mission types, and each one will have a variety of *Connection Points*. The *Connection Points* have attributes such as *Connection Type* and *Orientation* to define relation between *Connection Points* and *Components*. The *Connection Points* can only be attached to *Component* types by having same *Connection Type* attribute. The attributes of *Connection Points* are shown in Table 3.2.

If all *Components* support the same *Connection Type*, the resulting number of configurations will be extremely large if the number of connection points and components is also large. For example, if there are 5 components and 20 possible connection points, there are a possible 15504 configurations that would need to be evaluated, as computed from Equation (3.4). Creation and evaluation of large numbers of satellite configurations can be computationally time consuming because of "state explosion".

Both *Connection Type* and *Orientation* attributes create constraints on the number of components that than can be associated with a specific *Connection Point*. Therefore, the magnitude of the "state explosion" can be significantly decreased as constraints are applied. Only using one type of connection interface might be more suitable in situations where available computer resources and time constraints are not a limiting factor. In the end, we think that the approach presented in this research might be best used to create a database of all satellite configurations beforehand from which various programs evaluate those configurations for user selection.

In our approach, *Connection Type* and *Orientation* attributes are created for each *Component* which reduces the number of available component location options for each component in the satellite, thus reducing number of possible configurations. Depending on the function and operating environment, certain components should

Table 3.2: *Connection Point* Attributes

***Connection Point* Attribute**	**Description**
Id	A unique name for the *Connection Point*
Connection Type	Common mechanical and electrical interfaces that *Connection Points* and *Components* have. This approach forces *Connection Points* to have only one *Connection Type* one *Connection Point*
Orientation	The orientation that *Connection Point* supports (+X, -X, +Y, -Y, +Z or -Z) only one *Orientation* for each *Connection Point*
Local Rotation	Rotation matrix based on the orientation that a *Connection Point* supports. This matrix is used to find the mass moments of inertia of the rotated component
Global Rotation	Rotation matrix of the connection point with respect to satellite origin. This rotation is used to find the total MOI of the configured satellite and placement of models in the 3D viewer
Translation	Offset of connection point from the assembly origin that is used to find the total MOI of the configured satellite

only be attached to specific locations on the satellite structure to satisfy restrictions. For example, we can enforce the constraint that an earth sensor must be oriented towards the earth in order to function properly. Currently, both *Component* and *Connection Point* entities have a *Connection Type* and *Orientation* attributes that must be satisfied before assigning a component to a component location. We also assume that a connection location can only support one connection interface and orientation, on the other hand, a component can operate correctly in more than one orientation but only have one connection interface. The relationship between satellite components and connection locations is similar to the relationship between the USB devices and desktop computers. A USB thumb drive (PnP component) can be plugged into any USB outlet of the same size on a computer. On the other hand, a typical USB outlet (connection location) is fixed on the computer case and only allows a USB device to be plugged in one orientation. Each *Component* type such as *Sun sensor*, *Earth Sensor*, *Thruster*, etc. has different *Connection Types*, even if the actual component has the same connection interface with other types of components.

When selecting component locations on a satellite structure, it is likely that different size components can overlap or obstruct the view of each other. Each satellite structure has a fixed number of specific *Connection Points* on the panels of the structure. The distance between *Connection Points* is computed from the *Dimensions* attributes of each *Component* type. Defining *Connection Points* that only support certain types of *Components* gives us the flexibility of arranging *Connection Points* that don't let components overlap each other, however this must be done manually at this time. Resolving overlap and obstruction issues automatically is discussed further as a future work item in Chapter .

Based on mass, MOI, and CM of components and the coordinates of component locations that components are attached, a total MOI for the configured satellite is calculated using the parallel axis theorem. Assuming the moments of inertia (I_{cm}) are given by the component manufacturer or are measured, when the axis of rotation is parallel with an axis about which the MOI is known and this axis is displaced by

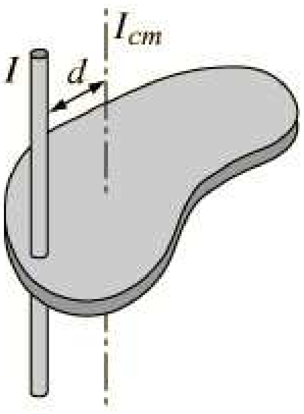


Figure 3.10: Parallel Axis Theorem

a distance d but is still parallel to the original axis, then the moment of inertia (I) through this axis, see Figure 3.10, can be calculated by

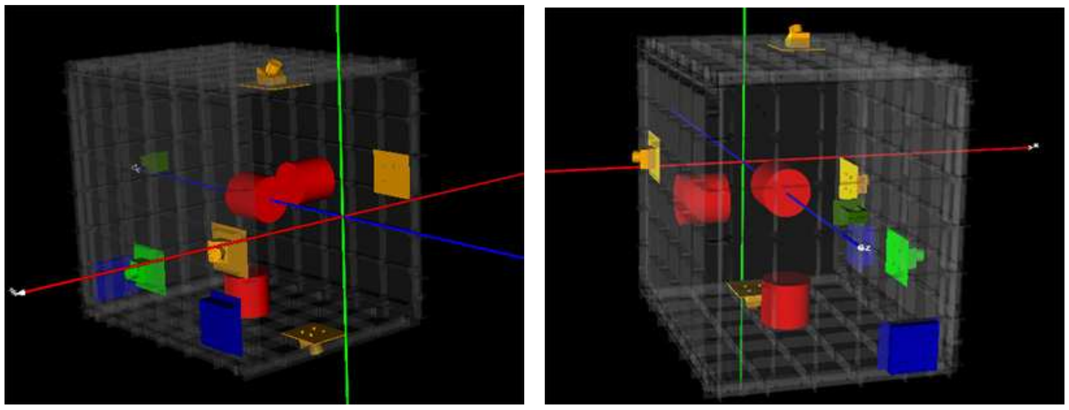
$$I_{parallel\ axis} = I_{cm} + Md^2 \tag{3.7}$$

or Equation can be expressed in 3D form as

$$\begin{aligned} I_{xx} &= \bar{I}_{x'x'} + M(d_y^2 + d_z^2) \\ I_{yy} &= \bar{I}_{y'y'} + M(d_x^2 + d_z^2) \\ I_{zz} &= \bar{I}_{z'z'} + M(d_x^2 + d_y^2) \\ I_{xy} &= \bar{I}_{x'y'} + M(d_x d_y) \\ I_{yz} &= \bar{I}_{y'z'} + M(d_y d_z) \\ I_{zx} &= \bar{I}_{z'x'} + M(d_z d_x) \end{aligned} \tag{3.8}$$

where M is the mass of the component and the prime indices are the known values around the component's center of mass.

It is assumed that CAD drawings, attributes such as mass, center of mass (CM), moments of inertia (MOI), etc. will be provided by the manufacturer of the components. If the drawings are not available, simple shapes like cube, cylinder, sphere etc. could be used to visualize the components on the satellites in a 3D environment. An



(a) (b)

Figure 3.11: Graphical Transparent Satellite Viewer
(a) and (b) show transparent satellite structure and color coded components from different angles

example 3D environment to view satellite configurations will be discussed in the next section.

3.6 Satellite Viewer - 3D Environment

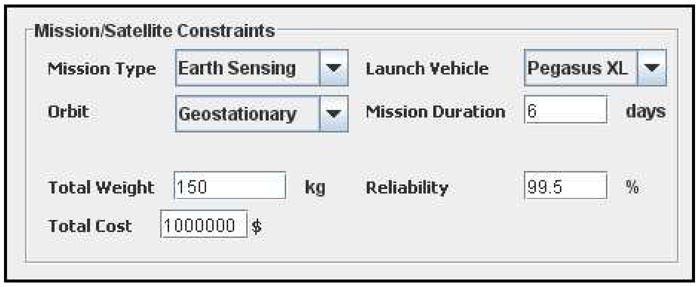
Because we expect to have 3D CAD models of every structure panel and component, we believe a satellite design tool should have a graphical 3D environment. An example of graphical 3D environment is shown in Figure 3.11. The graphical 3D environment should help the user examine and visualize satellite configurations from different angles by rotating the satellite on the screen. In this example, components are colored coded according to subsystems they belong to. For example, Attitude Determination and Control Subsystem (ADCS) components are colored "green", Power Subsystem components are colored "red" etc. The satellite structure is transparent so that the interior components can be easily viewed and the user can point out any existing inconsistencies in the configuration set up such as obstructions or overlaps. An example screen shot of current Satellite Viewer is shown in Figure 3.11.

3.7 Graphical User Interface (GUI)

GUIs were created to interface with users in viewing satellite configurations and calculating MOI, CM, total mass and, cost of configured satellites. Figure 3.12 is an example constraints GUI that allows users to input *Mission Type*, *Mission Duration*, *Orbit*, *Reliability*, *Weight* and *Cost* constraints. More constraints can be added to this incomplete list which has been created to test the approach. From the user input, three levels of constraints are created: *Satellite*, *Subsystem*, and *Component*. These constraints are used to reduce numbers of the satellite configurations. Through the GUI, the user not only inputs the constraints but can also access PDF files of available resources that exist in the warehouse. The user can select specific components for specific locations or leave their options open to evaluate a variety of configurations. Constraints levels and interfacing with the user are discussed further in Subsection 3.8.1.

Like PnP components, the panels can be used to create a variety of assembled satellite structures from different sizes (small, medium, large). An approach listing a limited variety of satellite structures assembled from those PnP plates could be developed to assist the user in selecting a satellite structure. *Mission Type* and *Orbit* are the two key factors that could be used determining the size of the satellite structure. An approach considering more than one satellite structure could be developed in future work. The selection of the satellite structure could also be automated later using a process that takes into account historical data. In this research, we did not test this concept, rather, we assumed there is only one satellite structure to which PnP components are attached during configuration process.

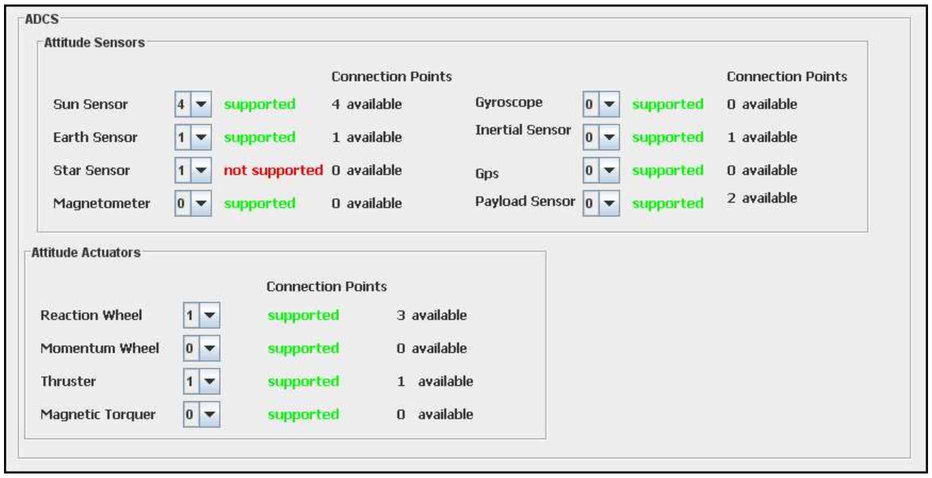
At this point, we will discuss one possible approach on how the user interface could be developed. The purpose of this GUI is to demonstrate an approach and aid in the testing of this constraint based approach. The logic for the GUI ensures that for each *Component* type there are enough *Connection Points* on the satellite structure to configure satellites. The user selects quantities for each different *Component* type



Mission/Satellite Constraints

Mission Type	Earth Sensing	Launch Vehicle	Pegasus XL	
Orbit	Geostationary	Mission Duration	6	days
Total Weight	150 kg	Reliability	99.5	%
Total Cost	1000000 $			

Figure 3.12: GUI-Mission and Satellite Constraints



ADCS

Attitude Sensors

			Connection Points				Connection Points
Sun Sensor	4	supported	4 available	Gyroscope	0	supported	0 available
Earth Sensor	1	supported	1 available	Inertial Sensor	0	supported	1 available
Star Sensor	1	not supported	0 available	Gps	0	supported	0 available
Magnetometer	0	supported	0 available	Payload Sensor	0	supported	2 available

Attitude Actuators

			Connection Points
Reaction Wheel	1	supported	3 available
Momentum Wheel	0	supported	0 available
Thruster	1	supported	1 available
Magnetic Torquer	0	supported	0 available

Figure 3.13: GUI-User Component Selection by Subsystem

to configure satellites, see Figure 3.13. The user is capable of viewing the current stock level of each *Component* type and can see the number of available *Connection Points* for those *Component* types. The user selects the maximum quantity for each *Component* type under the restriction of available *Connection Points*. The user is informed if they select component quantities from which no combination can be created, due to lack of connection locations on the selected structure or component availability in the warehouse. Additionally, the quantity of each *Component* in the library has to be equal to or smaller than the available *Connection Points* for that *Component* type so that valid combinations can be created. The availability of *Connection Points* is determined by matching attributes of each *Component* and *Connection Point*.

3.8 Methods Developed to Create PnP Satellites

As a reminder, the italicized term *Component* represents entities that only exist in the virtual library, and the term "Component" indicates a physical object in the warehouse. A *Component* is the highest level of entity in the OOD that all other entities inherits from. *Components* performing different functions grouped together form *Component* types such as *Sun Sensors*, *Earth Sensors*, *Reaction Wheels*, etc, as shown in Figure 3.14. The *Availability* attribute of a *Component* is the stock level of real components in the warehouse that are represented by "Components in the Library" in the same figure. The actual Sun Sensor 1 (SS1) belongs to *Sun Sensor* type, and there are four Sun Sensor 1 components in the warehouse.

3.8.1 Interfacing with the User.

Mission requirements, PnP components, and satellite structures are the three major elements of the user input in this constraint-based approach. The user must input these major elements in order to determine *Satellite*, *Subsystem*, and *Component Level Constraints*.

Satellite Level Constraints consist of mission specified constraints such as *mission type*, *mission duration*, *orbit*, *weight*, *cost*, and *reliability* etc. From these user inputs, *Satellite Level Constraints* are generated which will be needed to eliminate infeasible satellite configurations. Maximum cost, maximum mass, and minimum reliability of satellite configuration are examples of *Satellite Level Constraints* that satellite configurations which violate one or more of those constraints are eliminated. *Mission type*, *mission duration*, and *orbit* are the main factors in selecting a satellite structure that houses the PnP components.

Component and *Subsystem Level Constraints* can be derived from *Satellite Level Constraints*. For example, a propellant constraint for orbit correction, maintenance, and attitude control can be calculated using the mission duration and orbit information for creating a *Component Level Constraint*.

The SMAD approach, as discussed in Subsection 2.2.1, has methods to estimate mass, cost, power budgets, etc. of each subsystem with respect to the entire satellite

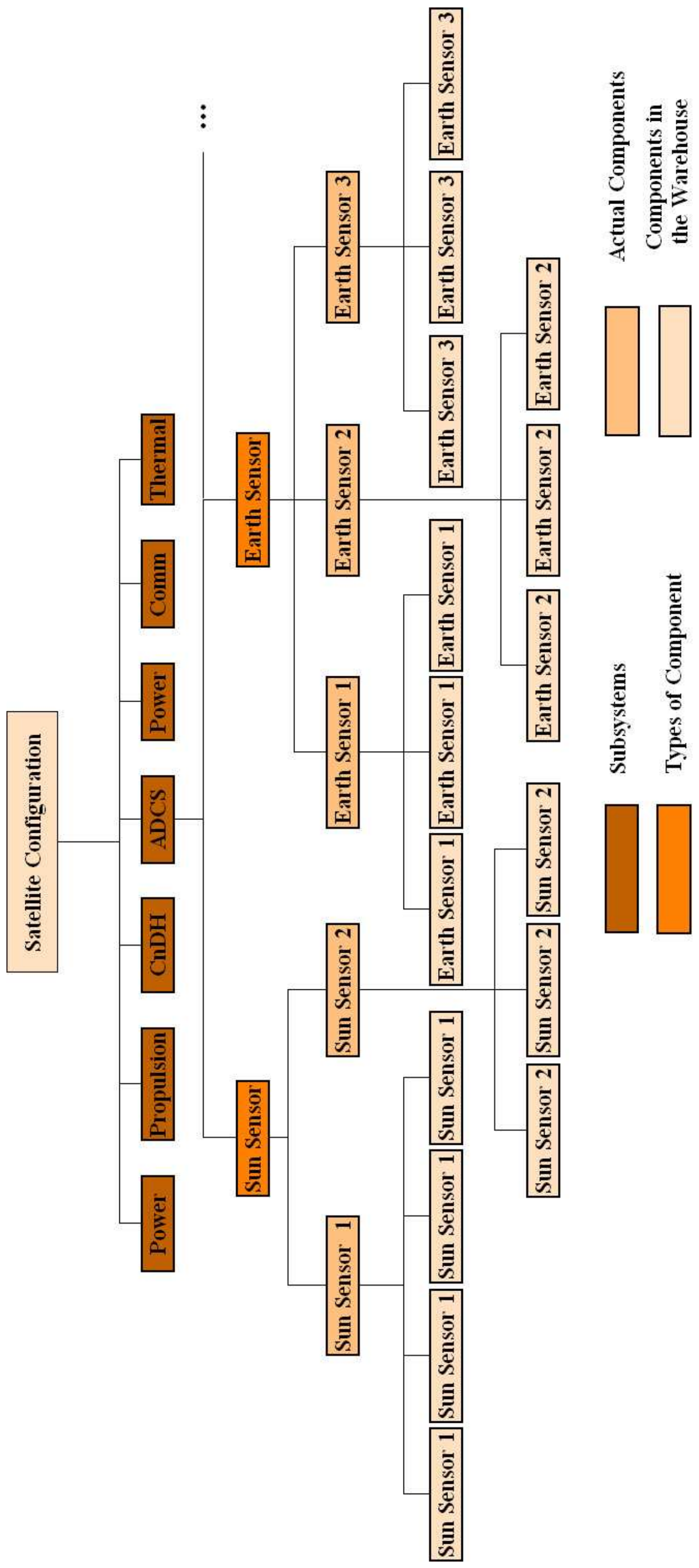


Figure 3.14: Component Hierarchy

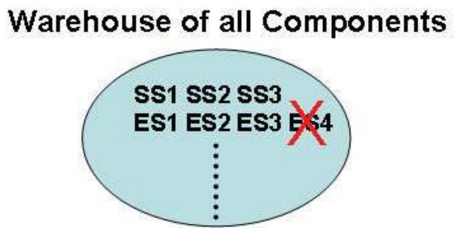


Figure 3.15: Example of a Component Level of Constraint Being Applied

based on satellite historical data. In the same way, applying subsystem percentages to satellite mass, weight, and cost constraints, *Subsystem Level Constraints* can also be created. From *Subsystem Level Constraints*, *Component Level Constraints* can also be generated. However, percentage estimates generated from small satellite historical data may lead to poor estimations because PnP Satellites are expected to have components and subsystems not specifically optimized for a specific mission. Generating and applying *Subsystem Level Constraints* are not evaluated in this research, but could easily be applied.

In addition to the estimated *Component Level Constraints*, the user inputs *Component Level Constraints* manually to prevent creation of invalid configurations based on the user's desires. For example, if a user decides that the accuracy of sun sensor must be higher than a specific value, they simply input an accuracy restriction as a *Component Level Constraint* such that all sun sensors in the library that exceed this constraint are not eligible for selection. Additionally, having the user select a specific or a group of components is a *Component Level Constraint* that is used in this research.

3.8.2 Selecting Valid Components.

A scenario has been created to explain and demonstrate the approach by creating a simple satellite (SimpleSat) consisting of a satellite structure and a simple ADCS subsystem composed of some sensor components like *Sun Sensor* (SS) and *Earth Sensor* (ES). The process of creating configurations and applying constraints to SimpleSat will make the explanation of

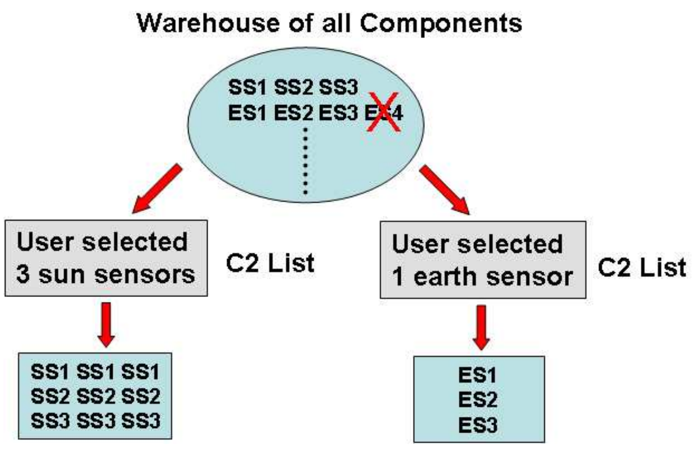

Warehouse of all Components
SS1 SS2 SS3
ES1 ES2 ES3 ES4
User selected 3 sun sensors
C2 List
User selected 1 earth sensor
C2 List
SS1 SS1 SS1
SS2 SS2 SS2
SS3 SS3 SS3
ES1
ES2
ES3

Figure 3.16: Example Creation of Component Combination List (C2 List)

this approach easier to follow, because only a limited number of *Components* and *Location Points* are used.

In the previous step, the minimum required inputs from the user collected through a GUI and various constraints are created and applied during the configuration process when applicable. First, depending on the user input and attributes of *Components* in the library, lists of valid *Components* are created. As shown in Figure 3.15, sets of *Sun Sensors* (SS1, SS2 and SS3) and *Earth Sensors* (ES1 ES2 ES3 ES4) are created from the library. Any *Component* which does not meet all the *Component Level Constraints* is eliminated from the list. A good example of a *Component Level Constraint* being enforced is shown in Figure 3.15 where ES4 is eliminated because the accuracy attribute of the ES4 component is not high enough to meet the user specified *Component Level Constraints* for *Earth Sensor Component* type.

3.8.3 Creation of the Component Combinations Lists (C2 Lists). Each *Component* that meets *Component Level Constraints* is used to create combinations of the desired number based on user input that specify quantities for each *Component* type. Component Combinations (C2s) composed of "like" (same manufacturer and model number) components that have same *Id* attribute are created, see Section 3.5.

For each *Component* type, one or more C2s are created if the user has decided to use that *Component* type in the configuration. As shown in Figure 3.16, three sun sensors and one earth sensor are selected, and C2 Lists for *Earth Sensor* and *Sun Sensor* are created.

To minimize the size of the C2 Lists, C2 entities are only created from "like" *Components*. As a future improvement, this restriction could be removed. Using different components can provide flexility, but it also can create some drawbacks like compatibility issues that might not be handled with simple logic. Components made by different manufacturers are likely to have different specifications. Thus, using different components requires more complex coding to handle differences between components to prevent conflicts and are not considered in this research.

Another reason to not allow dissimilar components that serve the same purpose is the number of combinations created could be extremely large, even when small quantities and varieties of components are used. For example, assume that we have ten sun sensors from different manufacturers, have quantities greater than three available, and want to attach all of these components in three locations on the satellite structure. In this case, allowing configurations that include different sun sensors would result in 10^3 different combinations just for possibly part of the attitude sensing capability of the satellite. However, if only "like" components can be used, we would only have 10 different combinations.

3.8.4 Creation of Combined Component Combinations List (C3 List).

To create a subsystem configuration, we first combine the C2 Lists and then assign component locations to those components. In the previous step, we created C2 Lists for each *Component* type based on the user input. The C3 List is created by selecting C2s from each of the C2 Lists and combining them with other C2s of different *Component* types to create a subsystem where component locations will be determined later. A C3 consists of Component Combinations (C2) from each created Component Combination List (C2 List) that defines a subsystem, as shown in Figure 3.17. The number

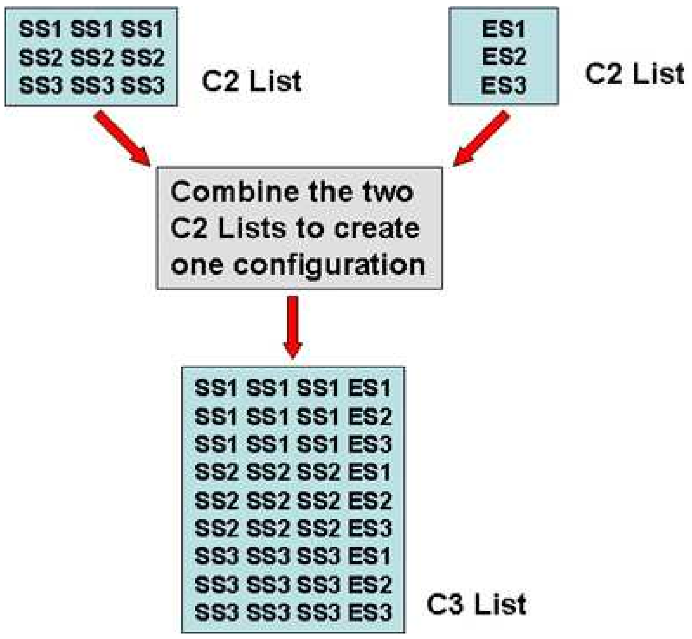


Figure 3.17: Example Creation of Combined Component Combination List (C3 List)

of C3s is directly related to number of C2s in each C2 List. For example, if we have ten C2 choices for sun sensor component, five C2 for earth sensor components, and ten for thruster components, the total number of options for ADCS will be $10(5)10 = 500$, refer to previous discussions on the multiplication principle in Section 3.3.

3.8.5 Removing Combinations that Violate Subsystem Level Constraints.

Each C3 in Combined Component Combination (C3) Lists describes components in a subsystem without locations, *Subsystem Level Constraints* are applied to C3 Lists to eliminate configurations in which a C3 violates any cost, weight, etc constraints. The *Satellite Level Constraints*, like reliability and launch vehicle (static/dynamic envelope), can only be applied after a complete satellite configuration has been created. However, *Subsystem Level Constraints* can be applied in this level, as discussed Subsection 3.8.1. The number of satellite configurations will increase exponentially as the number of C3s increases. Applying *Subsystem Level Constraints* as soon as possible reduces the effect of combinatorial explosion in the component configurations. In the example SimpleSat scenario, we are creating satellite configurations consisting of only

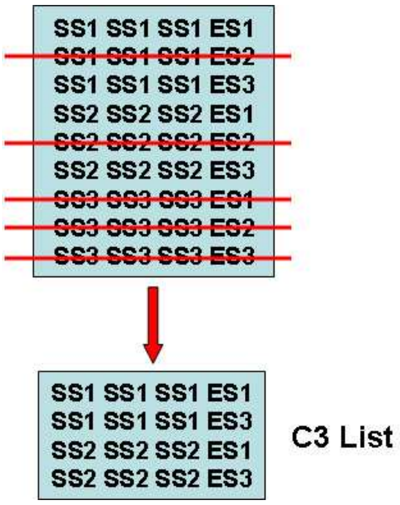
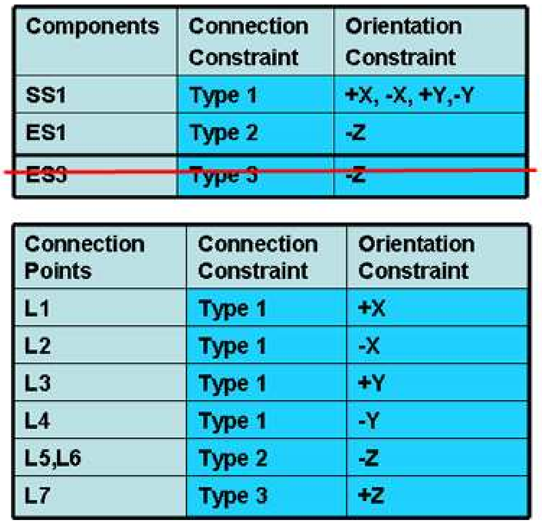


Figure 3.18: Example of Removing Invalid Combinations Depends on User Constraints

ADCS subsystem to explain the approach. Therefore, *Satellite Level Constraints* such as total mass and total cost can be applied as *Subsystem Level Constraints* only because the subsystem configuration also represents satellite configuration in this special case. In the Figure 3.18, an example of an eliminated configuration due to exceeding the mass and cost constraints is shown.

3.8.6 Removing Invalid Combinations that Violate Available Connection Points. At this point, the C3 Lists have subsystem level combinations that are not valid because of lacking *Connection Points* on the satellite structure for components in the combinations. As explained in Subsections 2.3.5 and 2.3.8, the user selected satellite structure has a limited number of *Connection Points*. As discussed in Section 3.5, a *Connection Point* can only support one *Connection Type* and one *Orientation*, on the other hand, a *Component* can have more than one *Orientation* but only one *Connection Type*.

The same type of *Component* may have different *Connection Types* as in computer systems, a mouse can have either USB or DB-9 serial connector. *Components* can only be attached to *Connection Points* of which *Connection Types* match. Any particular *Component* for which there are not enough *Connection Points* available



Components	Connection Constraint	Orientation Constraint
SS1	Type 1	+X, -X, +Y,-Y
ES1	Type 2	-Z
~~ES3~~	~~Type 3~~	~~-Z~~

Connection Points	Connection Constraint	Orientation Constraint
L1	Type 1	+X
L2	Type 1	-X
L3	Type 1	+Y
L4	Type 1	-Y
L5,L6	Type 2	-Z
L7	Type 3	+Z

Figure 3.19: Example of Component and Connection Point Properties

on the selected structure results in elimination of any C3 that includes that particular *Component*. For example, as shown Figure 3.19, ES3 can only be connected to *Connection Points* that support a Type 3 connector and -Z orientation. On the other hand, L7 *Connection Point* has a Type 3 connector, however this *Connection Point* supports only a +Z orientation that makes L7 invalid for SS3. Because the user selected satellite structure does not have the designated connector required for ES3, C3s that have ES3 are not valid and are eliminated from C3 List as shown in Figure 3.20.

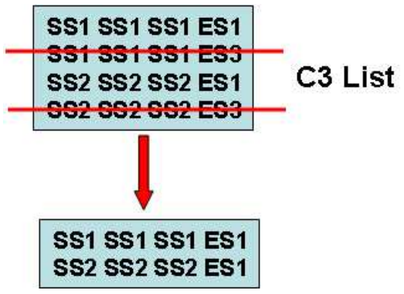


Figure 3.20: Example of Removing Invalid Combinations that Depend on Available Connection Points

3.8.7 Creating C3 Location Lists. The elimination of C3s as described in SubSection 3.8.6 ensures that there are enough *Connection Points* on satellite structure. In the same way that Component Combinations (C2) are created from components and user inputs, combinations of *Connection Points* for each C3 are created from available *Connection Points* on the selected satellite structure.

So far, C2 Lists have been combined to form Combined Component Combinations (C3s) as explained in Subsection 3.8.4. Only C2 Lists that consist of "like" components for the user selected component quantity are allowed. For each C3, the number of "like" components in C2 and available *Connection Points* are used to create lists of combinations of C2 Locations. Different from creating C2s, these C2 Locations Lists incorporate the requirement that *Connection Points* are used only once. After creating all of the C2 Locations Lists, the C2 Locations Lists are combined to create the C3 Location List in the same way C3 Lists are created as described in SubSection 3.8.4. The next step is to create subsystem combinations by combining the C3 List and C3 Location Lists. An example of previously described processes is shown in Figure 3.21.

3.8.8 Creating Subsystem and Satellite Configurations. In the previous step, for each C3 one C3 Location List consisting of various C3 Locations is created. C3s and C3 Location Lists are combined to create subsystem configurations. Combining those C3s with corresponding C3 Locations create numerous subsystem configurations if the number of available locations are higher than number of components in C3s. From each C3 and C3 Location List, at least one subsystem configuration is created.

All of the examples given so far are only for the ADCS components. Only after configuring other satellite subsystems like Command and Data Handling (CnDH), Communication, Power etc. can we combine subsystems and create various satellites. Because this research effort is focused on evaluating a new approach, only ADCS components will be discussed.

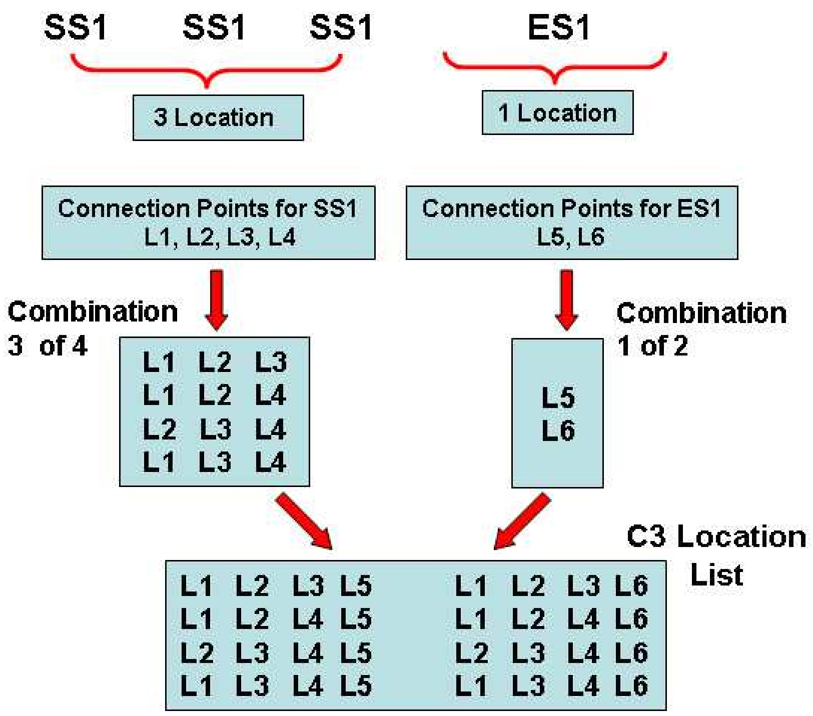


Figure 3.21: Creating C3 Location Lists

3.8.9 Elimination of Satellite Configurations that Violate Satellite Constraints.

The *Satellite Level Constraints* like total mass, total cost, and total power can be converted to *Subsystem Level Constraints* as explained in Subsection 3.8.1 and can be applied right after C3s are created for each subsystem. However, calculations like MOI and CM require all component locations for a complete satellite configuration be known. Therefore, *Satellite Level Constraints* such as MOI, CM, Static/Dynamic Envelope, and reliability are only applied after complete satellite configurations are created.

3.8.10 Sorting Satellites Configurations. With a large variety of components in the warehouse and multiple location points, especially if they are of the same connection type, it is very likely that an exceedingly large number of satellite configurations could be created from which it would be daunting for the user to decide on the best configuration. In this section, we look at different ways to find the best configuration that satisfies the users need from possibly thousands to millions of valid combinations. Because the user may want the lowest mass or least expensive satellite configuration that meets the mission requirement constraints, a sorted list can help. The user should be able to sort based on criteria such as weight, cost, size or combinations of these variables and possibly with corresponding weightings when more than one criteria is desired.

Additionally, flight history of components may be a determining factor in selecting specific components. Satellite configurations may be sorted according to the number or percentage of components that have a good record of space qualification and possibly success in a particular configuration to allow user to find the most reliable configurations.

In our research, we also found that sorting satellite configurations resulted in large numbers of configurations being grouped together. For example, sorting by cost we found that the ten least expensive configurations were nearly identical. Therefore, we evaluated a modification of our sorting approach. When satellite configurations are created from a fixed set of known components, some satellites may be very similar to each other. Likewise, there are components in the library having very similar specifications that may result in creating "nearly identical" satellite configurations. An approach of normalizing the attributes of the satellite and eliminating "nearly identical" satellite configurations can significantly reduce the number of configurations from which the user can select from.

Figure 3.22 displays 4800 satellite configurations based on their reliability, weight and cost. Each dot represents a satellite configuration, and the encircled dots represent dissimilar satellite configurations based on the user "criteria" input. The "criteria" is a measure of distance (as defined below) between configurations, and its value ranges between 0 and 1.

Before a measure of distance between configurations can be determined, the Cost, Weight and Reliability attributes of satellites must be normalized into a range of 0 and 1. Normalization is required to provide equal weighting between satellite

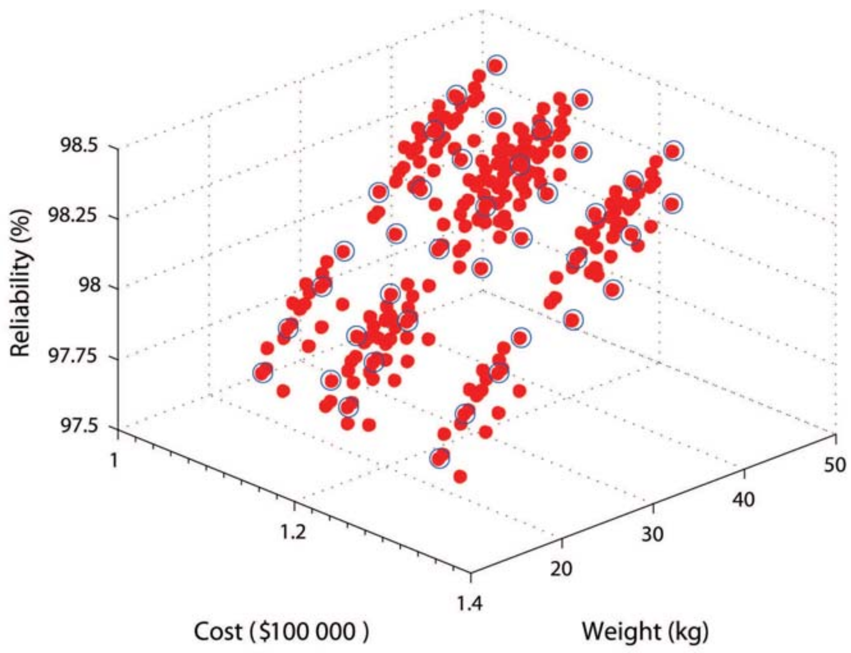


Figure 3.22: Example of Selection of Satellites Based on Reliability, Cost and Weight Criteria

attributes when determining the similarity of two satellites, if equal weighting is desired. The following equation is used to calculate a normalized value between 0 and 1 from a set of data $\{d1,\ d1,\ d2,\ d3 \ldots dn\}$

$$\delta = \frac{d - d_{\min}}{d_{\max} - d_{\min}} \tag{3.9}$$

where δ is the normalized value, d is the value to be normalized, $d_{\min}$ and $d_{\max}$ are the minimum and maximum values in the set of data, respectively.

Representing satellite configurations as points in three dimensional space using mass, cost and reliability properties and estimating the distance between two satellite can be calculated using the Euclidean norm

$$d = |\mathbf{x} - \mathbf{y}| = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2} \tag{3.10}$$

where d is the distance between points x and y in Euclidean n-space of reals also written as $\mathbb{R}^n$ [10].

For example, we can estimate the similarity of two satellite P and Q, where $P = (p_x, p_y, p_z)$ and $Q = (q_x, q_y, q_z)$ with attributes of mass, cost, and reliability from

$$d = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2} \tag{3.11}$$

where d can be thought of as a measure of similarity between the two satellites. Similar satellites have the same or close values of d.

IV. Results and Analysis

Several test scenarios were created to analyze the approach developed in this research by evaluating the effect of various constraints in reducing the number of satellite configurations created. In the first section of this chapter, the correlation between the number of components and the number of configurations created is demonstrated, and the effect of applying various constraints to number of component configurations is discussed. In the second section, results of the "constraint based" approach developed to minimize "combinatorial explosion" when creating PnP satellites is discussed.

4.1 Creating Component Configurations

The purpose of this section is to demonstrate different cases in which small numbers of *Components* and *Connection Points* can result in very large numbers of configurations. Then, by applying a constraint based approach, we reduce the number of configurations to assist the user in rapidly finding configurations that satisfies user's needs in priority order. In this section, only one *Component* type will be used to create configurations, and in subsequent steps, new constraints will be introduced resulting in a cumulative effect on the total number of valid configurations. We assume only ten sun sensors are available in the warehouse. The applied constraints and resulting number of available locations on the satellite structures are presented in the tables.

4.1.1 Configurations Created Using "various" Components. As explained in Subsection 3.3, numerous combinations can be created from only a few *Components* and *Connection Points* in the virtual repository if we allow combinations to have mixed *Components*, as opposed to "like" components. For example, in a scenario where we want to create configurations composed of five *Components* from a selection of ten *Components* and nine *Connection Points*, we can create 16.8 million configurations, using Equations (3.6) and (3.4). As explained in Section 3.8, we first create *Component* Combinations (C2) and then C2 Location configurations from C2s to allocate locations to components. As summarized in Table 4.1, different from the method created in this approach, various *Component* types are used to create *Com-*

Table 4.1: Scenario Created for Using “various” *Components* to Create Configurations

Component Level Constraints		no
Availability Constraints		no
Connection Type Constraints		no
Orientation Constraints		no
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	10	9

ponent combinations (C2) to demonstrate the effect of using various components in configurations. No constraints such as *Connection Type*, *Orientation* etc. are applied. In the case of using various components in configurations, the number of created *Component* combinations is extremely large when compared to the number of *Component* configurations created using an approach that uses “like” *Component* only.

Table 4.2: Sample of 21 of 70 Sun Sensor Combinations

SS1	SS1	SS1	SS1	SS1	SS2	SS4	SS5	SS2	SS3	SS3	SS4
SS1	SS1	SS1	SS2	SS1	SS2	SS5	SS5	SS2	SS3	SS3	SS5
SS1	SS1	SS1	SS3	SS1	SS3	SS3	SS3	SS2	SS3	SS4	SS4
SS1	SS1	SS1	SS4	SS1	SS3	SS3	SS4	SS2	SS3	SS4	SS5
SS1	SS1	SS1	SS5	SS1	SS3	SS3	SS5	SS2	SS3	SS5	SS5
SS1	SS1	SS2	SS2	SS1	SS3	SS4	SS4	SS2	SS4	SS4	SS4
SS1	SS1	SS2	SS3	SS1	SS3	SS4	SS5	SS2	SS4	SS4	SS5
...	...	...	...	...	...	...	...	...	...	...	...

Table 4.3: Configurations Created from 5 “like” *Components* using 4 at a time

SS1 SS1 SS1 SS1	SS4 SS4 SS4 SS4
SS2 SS2 SS2 SS2	SS5 SS5 SS5 SS5
SS3 SS3 SS3 SS3	

To demonstrate the effect of using various *Component* on the number of *Component* combinations, a sample of 21 C2s of total 70 combinations, see Equation (3.5), created from 5 various *Component* (SS1, SS2, SS3, SS4, SS5) for 4 *Connection Points* (L1, L2, L3, L4) is shown in Table 4.2. The large difference, 70 versus 5, in the number of *Component* combinations can clearly be seen when we compare Tables 4.3 and 4.2. Because we chose to use various *Components* of different brand or specifications to create configurations, the number of end configurations increases with the number of *Components* selected, as seen in Figure 4.1. The number of *Component* combinations and *Component* locations for each *Component* combination determine the total number of configurations. Figure 4.1 illustrates the growth in the number of configurations increases approximately as a function of $n!$ as seen in Equation (3.5) as the number of selected *Components* n increases.

The number of configurations created is many millions which makes it almost impossible to manage because the user would be overwhelmed trying to determine which configuration to choose. Creating configurations by combining various *Components* is not considered a good approach to follow because of the factorial increase of the high number of configurations and more importantly the complexity of handling components with different brands and specifications.

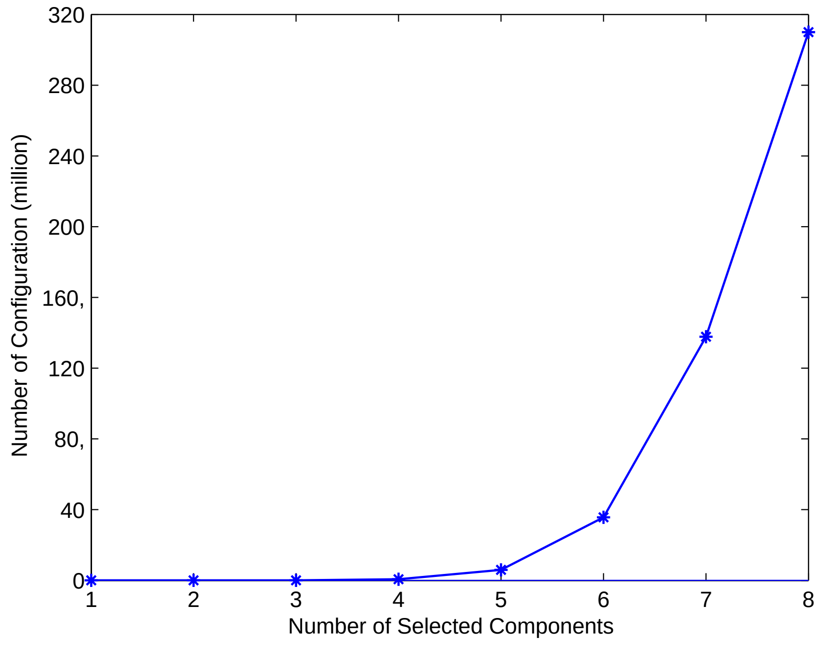


Figure 4.1: Number of Configurations versus Number of Selected *Components* where there are 10 available *Components* and 9 *Connection Points*

4.1.2 Configurations Created Using "like" Components. In previous subsection we discussed the effect of not having the restriction of allowing "like" *Components* only. This subsection will discuss the effect of using "like" *Components* which results in a more manageable number of configurations. We conclude that allowing only "like" *Components* as a restriction is a better methodology than allowing "various" *Components* to follow.

By only allowing "like" *Components*, the number of end configurations reduces from millions to a more manageable tens of thousands for the scenario given in Subsection 4.1.1. *Component* combinations from five *Components* is shown in Table 4.3. Comparing Tables 4.3 and 4.2, we can see that there are 65 more *Component* combinations created when various *Components* are used in configurations. In the approach that we created in this research, the number of *Component* combinations does not change as the number of selected *Components* increases. The number of *Component* combinations is fixed and equals to the number of *Components* in the warehouse because only "like" *Component* combinations are allowed. A scenario created to test this is shown in Table 4.4. In the scenario, 10 "different" types of sun sensors and 16 locations are available to create configurations. The number of configurations from 10 sun sensors and 16 locations as the number of selected *Components* increases is demonstrated in Figure 4.2. Because the number of *Component* combinations is fixed, the number of *Component* locations is the key variable that determines the number of total configurations. In this case, the number of *Component* locations can be calculated using Equation (3.4). This calculation results in the number of satellite configurations being bounded, as shown in Figure 4.2.

In following subsections, we will introduce constraints to the previous scenario and discuss the effect of constraints on the created configurations.

4.1.3 Reduction in Configurations Due to Applying Component Level Constraints. As discussed in Subsection 3.8.2, any *Component* that does not meet the *Component Level Constraints* is eliminated. We assume two sun sensors in the

Table 4.4: Scenario Created for Using “like” *Components* to Create Configurations

Availability Constraints		no
Connection Type Constraints		no
Orientation Constraints		no
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	10	16

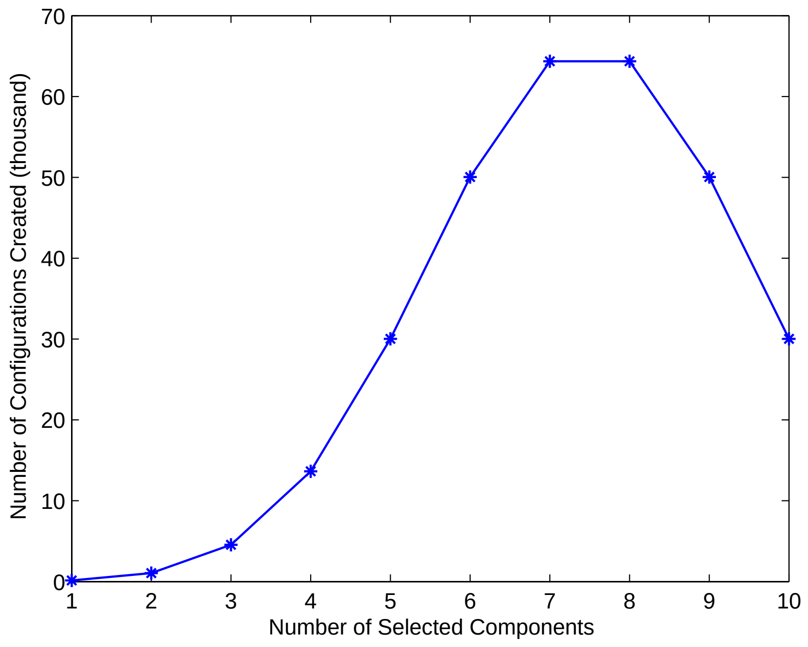


Figure 4.2: Configurations Created Using “like” *Components*

warehouse have accuracy values lower than user’s accuracy constraint. Therefore, two *Components* are eliminated by this *Component Level Constraint* and the number of available *Components* in the warehouse for configuration decreases from ten to eight, see Table 4.5. The number of *Component* locations did not change for each component combination. However, because the number of *Component* combinations decreases, the total number of configuration also decreases, respectively. Additionally, as the number of *Component Level Constraints* increases, there is a resulting decrease in the number of the configurations because more *Components* become invalid for configuration and are eliminated. In Figure 4.3, the result of reduction in the number of component configuration with regard to number of *Components* used in the configurations is shown. The maximum number of configuration decreased from 64 thousand to 50 thousand when we apply *Component Level Constraints* as described in the previous scenario. Because the number of *Component* combinations is fixed, the number of *Component* locations still has a driving effect on determining the number of total configurations that can be calculated using Equation (3.4).

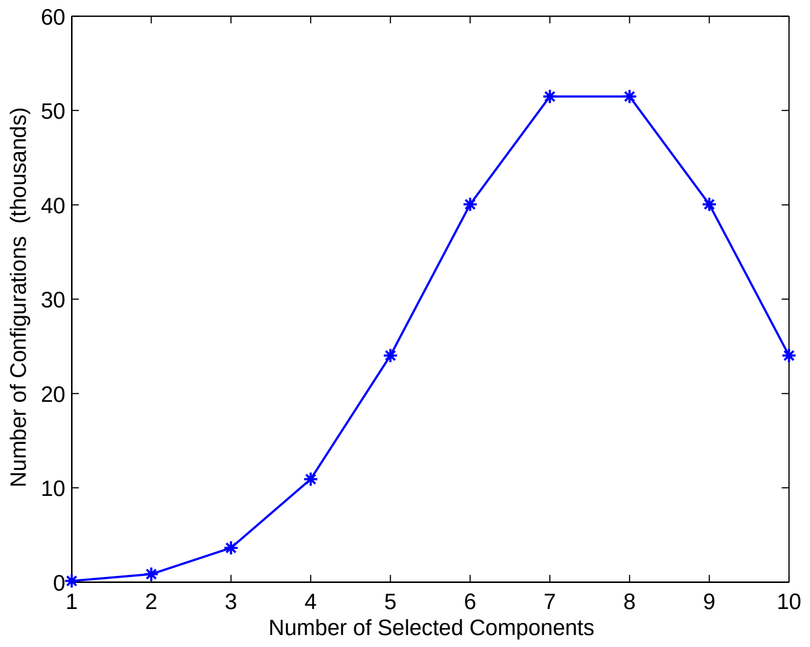


Figure 4.3: Configurations Created Component Level Constraints Applied

4.1.4 Reduction in Configurations Due to Applying Availability Constraint. In previous sections, we discussed that the created *Component* configurations are fixed and assumed there is an unlimited quantity of components in the warehouse. However, limiting quantities in the warehouse will constrain the resulting number of configurations.

In the scenario summarized in the Table 4.6, each *Component* has limited a quantity which varies between 1 to 9 (inclusive). The quantities of the *Components*

Table 4.5: Scenario Created for Applying Component Level Constraints to Create Configurations

Component Level Constraints		yes
Availability Constraints		no
Connection Type Constraints		no
Orientation Constraints		no
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	8	16

are determined by Gaussian probability distribution function (PDF) that was explained in Section 3.2. Because there are no component of a quantities of ten, no configuration composed of ten components is created, see Figure 4.4. In this case, the number of *Component* combinations is not fixed as the number of selected *Components* increases because the availability of components is limited. If the number of *Component* combinations was fixed, the number of locations would govern the total number of configurations and Figure 4.4 would peak when the number of selected *Component* is eight. As seen in the same figure, the number of configurations reaches its peak value when the number of selected components is six because there are not *Components* quantities higher than six in the virtual warehouse. The number of *Component* locations still depends on the selected number of *Components* (1-9) and available locations (16). Even though the number of *Component* configurations varies, the total number of configurations decreases as number of selected *Components* comes closer to the number of available locations.

From Figure 4.4, we can see that the maximum number of configurations decreases from 50 to 20 thousand. Because we have limited the number of available components, the user can not change the availability of a components directly but can become aware of this constraint through a GUI as discussed in Section 3.7.

4.1.5 Reduction in Configurations Due to Applying Connection Type Constraints. In previous subsection, we still had a maximum of 20 thousand con-

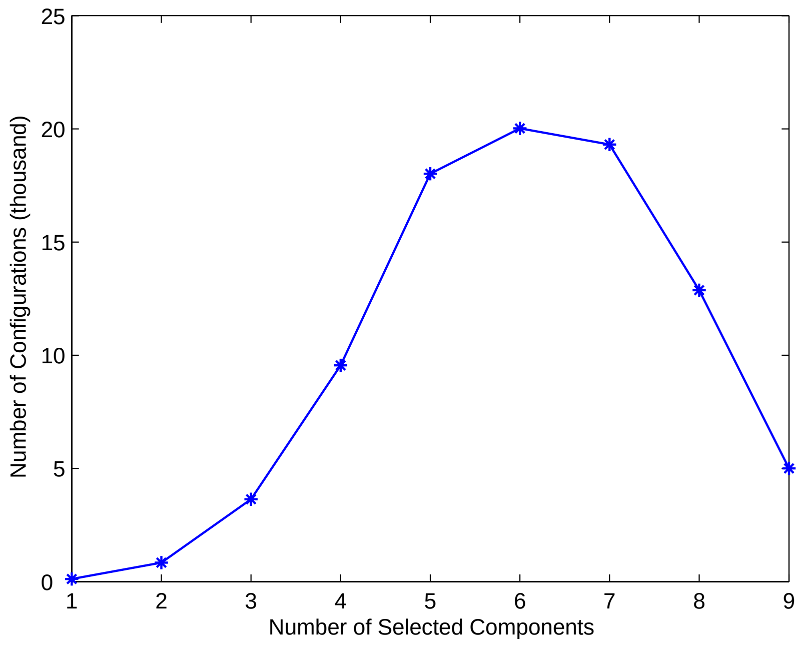


Figure 4.4: Configurations Created Availability Constraints Applied

figurations from only 8 *Components* and 16 *Component* locations. The main reason for the large number is that having too many component locations for components to be connected, in other words, there is only one *Connection Type* and any *Component* can be assigned to any *Connection Point*. Therefore, *Connection Points* on which components can be connected to will be enforced, or in other words, not every *Connection Point* is valid for all components. The *Connection Type* attributes of *Components* creates a constraint on the number of available *Component* locations that can be associated with specific *Component*. In the scenario summarized in Table 4.7, some of the *Connection Points* for the *Components* are eliminated due to the previously discussed constraints on *Connection Type* attributes of the *Components* and *Connection Points*, which was also explained in Section 3.5. Of the 16 *Connection Points* on the satellite structure, only 12 of them are designated for *Sun Sensors*. Because the number of end configurations is directly related to the *Connection Points* combinations, the maximum number of configurations is decreased significantly from

Table 4.6: Scenario Created for Applying Availability Constraints to Create Configurations

Component Level Constraints		yes
Availability Constraints		yes
Connection Type Constraints		no
Orientation Constraints		no
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	8	16

20 to 2 thousand as shown in Figure 4.5. Just this simple limitation on the type of components that can be connected to a given location helps us considerably reduce the number of end configurations to manageable levels.

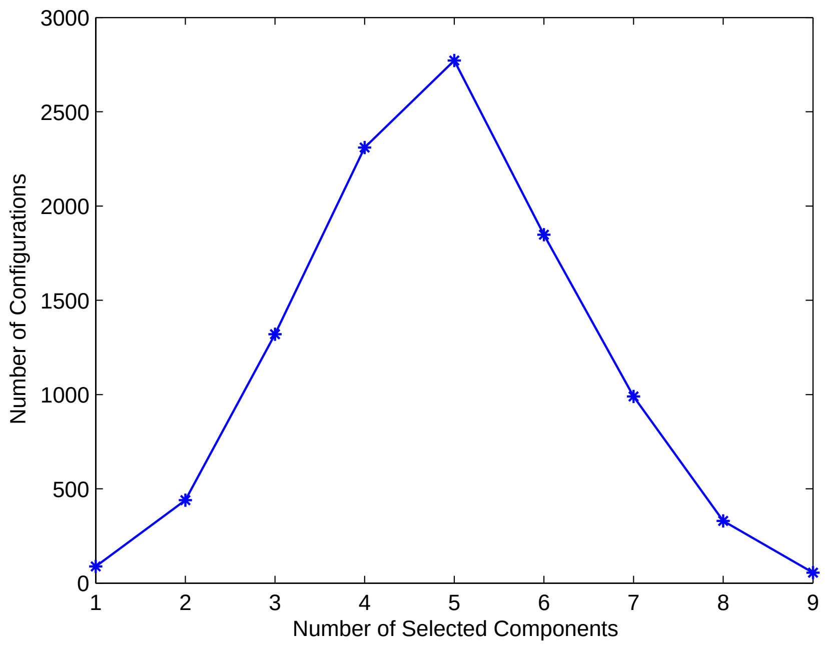


Figure 4.5: Number of Configurations Created Connection Type Constraints Applied

4.1.6 Reduction in Configurations Due to Applying Orientation Constraints.

In addition to *Connection Type Constraints*, *Orientation Constraints* are also created and tested. Each *Component* type has *Orientation* attributes based on their func-

Table 4.7: Scenario Created for Applying Connection Type Constraints to Create Configurations

Component Level Constraints		yes
Availability Constraints		yes
Connection Type Constraints		yes
Orientation Constraints		no
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	8	12

Table 4.8: Scenario Created for Applying Orientation Constraints to Create Configurations

Component Level Constraints		yes
Availability Constraints		yes
Connection Type Constraints		yes
Orientation Constraints		yes
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	8	7

tion and operating environment as explained in Section 3.5. Applying *Orientation Constraints* results in the elimination of *Connection Points* that result in a reduction in the number of available Connection Points. In the scenario summarized in Table 4.8, there are only seven valid *Connection Points* for *Components.* The Figure 4.6 shows the number of configurations created after orientation constraints are applied. The number of components selected can not be higher than available number of *Connection Points* that is seven as seen in the Figure 4.6. The maximum number of configurations decreased from two thousand to two hundred. Applying *Orientation Constraints* with the *Connection Type Constraint* decreases the total number of configurations significantly.

4.2 Combinatorial Reduction of Number of Satellite Configurations

In the previous Section, we focused on configurations created from a single type of component. In order to demonstrate the capability of the overall approach,

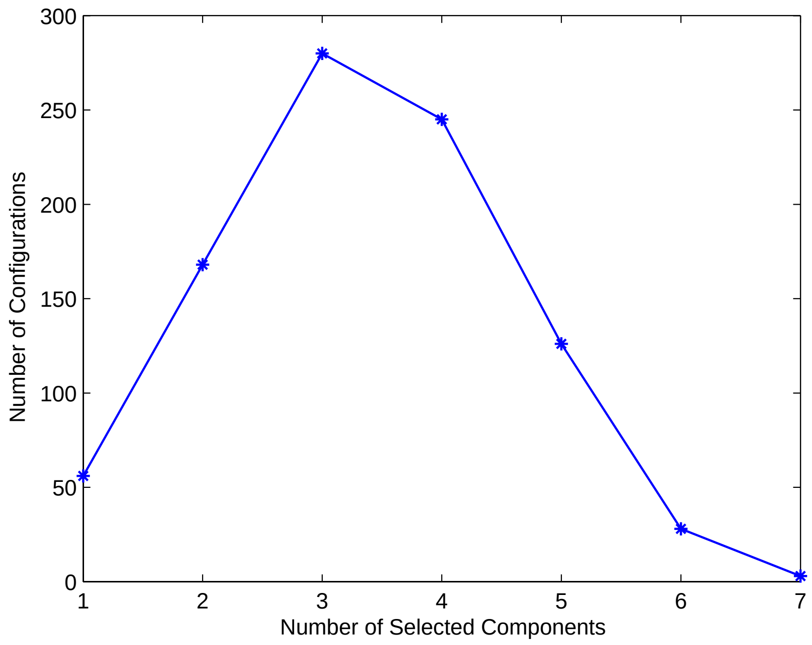


Figure 4.6: Configurations Created Orientation Constraints Applied

various scenarios are created and summarized in Tables 4.9 through 4.15. When all required components are used to create PnP Satellite, the number of configurations created without applying constraints is larger than several billion. Therefore, to keep the number of configurations at an manageable level, only some of the PnP Satellite components such as the sun sensors, earth sensors, thrusters, payloads, and reaction wheels are selected. The numbers of selected components is 4, 1, 4, 2, and 3, respectively. The valid number of *Components* in the virtual warehouse and *Connection Points* on the satellite structure were chosen arbitrarily. Each scenario has new applied constraints and the effect of those applied constraints on the number of valid *Components* and *Connection Points* is highlighted in the tables. The outcome of the applied constraints are shown as a number of satellite configurations at the end of the tables.

In the scenario depicted in Table 4.9, no constraints are applied when configuring satellites. There are 16 available *Connection Points* designated to all *Component* types in the configurations. The total number of configurations created is extremely large when no constraints are applied as represented in the same table.

In the scenario depicted in Table 4.10, only *Component Level Constraints* are applied (see Section 3.7). As a result of applying *Component Level Constraints*, two sun sensors, two reaction wheels, and one payload are eliminated. The number of configurations decreases by almost 50% when compared to the previous scenario. More user constraints, like *Component Level Constraints*, results in a significant decrease in the number of satellite configurations. *Component Level Constraints* by themselves can not decrease the total number of combinations to a manageable number because of the numerous components in warehouse.

In the scenario depicted in Table 4.11, *Availability Constraints* are applied as discussed in Subsection 4.1.4. As a result of applying *Availability Constraints*, two sun sensors and three reaction wheels are eliminated. The total number of configurations decreases almost 50% when compared to the previous scenario. Overall, applying both the *Availability Constraints* and *Component Level Constraints* still can not decrease the total number of combinations to a manageable number.

In the scenario depicted in Table 4.12, *Connection Type Constraints* are applied as discussed in Section 3.5. As a result of applying *Connection Type Constraints*, components are allowed to be attached only to designated locations on the satellite structure. As shown in the Table 4.12, the available *Connection Points* for components decreases significantly. Notice that the sum of available *Connection Points* for *Components* is 16. The total number of configurations decreases from 3.6 trillion to 18 thousand when we apply *Connection Type Constraints.*

In the scenario depicted in Table 4.13, *Orientation Constraints* are applied as discussed in Section 3.5. As a result of applying *Orientation Constraints*, the number of designated locations on the satellite structure for earth sensor and payload decreases to 1 and 2, respectively. The total number of configurations decreased from 18 thousand to 3 thousand when we apply these example *Orientation Constraints.*

Table 4.9: Satellite Configurations Created by Introducing no Constraint

Component Level Constraints		no
Availability Constraints		no
Connection Type Constraints		no
Orientation Constraints		no
Satellite Level Constraints		no
Similar Configuration Elimination		no
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	10	16
Earth Sensor	5	16
Thruster	4	16
Payload	5	16
Reaction Wheel	10	16
Number of Satellite Configurations		1.51351E+13

Table 4.10: Satellite Configurations Created by Introducing Component Level Constraints

Component Level Constraints		yes
Availability Constraints		no
Connection Type Constraints		no
Orientation Constraints		no
Satellite Level Constraints		no
Similar Configuration Elimination		no
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	8	16
Earth Sensor	5	16
Thruster	4	16
Payload	4	16
Reaction Wheel	8	16
Number of Satellite Configurations		7.74918E+12

Table 4.11: Satellite Configurations Created by Introducing Availability Constraints

Component Level Constraints		yes
Availability Constraints		yes
Connection Type Constraints		no
Orientation Constraints		no
Satellite Level Constraints		no
Similar Configuration Elimination		no
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	6	16
Earth Sensor	5	16
Thruster	4	16
Payload	4	16
Reaction Wheel	5	16
Number of Satellite Configurations		3.63243E+12

Table 4.12: Satellite Configurations Created by Introducing Connection Type Constraints

Component Level Constraints		yes
Availability Constraints		yes
Connection Type Constraints		yes
Orientation Constraints		no
Satellite Level Constraints		no
Similar Configuration Elimination		no
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	6	4
Earth Sensor	5	2
Thruster	4	4
Payload	4	3
Reaction Wheel	5	3
Number of Satellite Configurations		18000

Table 4.13: Satellite Configurations Created by Introducing Orientation Constraints

Component Level Constraints		yes
Availability Constraints		yes
Connection Type Constraints		yes
Orientation Constraints		yes
Satellite Level Constraints		no
Similar Configuration Elimination		no
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	6	4
Earth Sensor	5	1
Thruster	4	4
Payload	4	2
Reaction Wheel	5	3
Number of Satellite Configurations		3000

In the scenario depicted in Table 4.14, satellite configurations whose total weight is larger than 60 kg and cost is higher than \$160,000 are eliminated. The area bounded in the lower left hand corner by dashed line in Figure 4.7 contains valid satellite configurations based on the *Satellite Level Constraints* where each "+" represents a valid configuration.

In the scenario depicted in Table 4.15, satellite configurations are selected based on the distance criteria 0.08 to reduce the number of satellite to manageable levels as explained in Subsection 3.8.9 and computed from Equation (3.10). The selected satellite configurations are encircled based on the summarized scenario in Table 4.15. The end number of configurations is decreased from hundreds to tens making it a more manageable number of satellite configurations.

Figure 4.9 shows the case specific overall combinatorial reduction in the number of satellite configurations based on the different constraints and methods. Case numbers 1 to 7 represent previously discussed scenarios as summarized in Tables 4.9 through 4.15. The most significant decrease in the number of satellite configurations occurs in Case 4 which is application of Connection Type constraints. The number

Table 4.14: Satellite Configurations Created by Introducing Satellite Level Constraint

Component Level Constraints		yes
Availability Constraints		yes
Connection Type Constraints		yes
Orientation Constraints		yes
Satellite Level Constraints		yes
Similar Configuration Elimination		no
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	6	4
Earth Sensor	5	1
Thruster	4	4
Payload	4	2
Reaction Wheel	5	3
Number of Satellite Configurations		431

of end configurations decreases from several millions to tens by the approach created from this research.

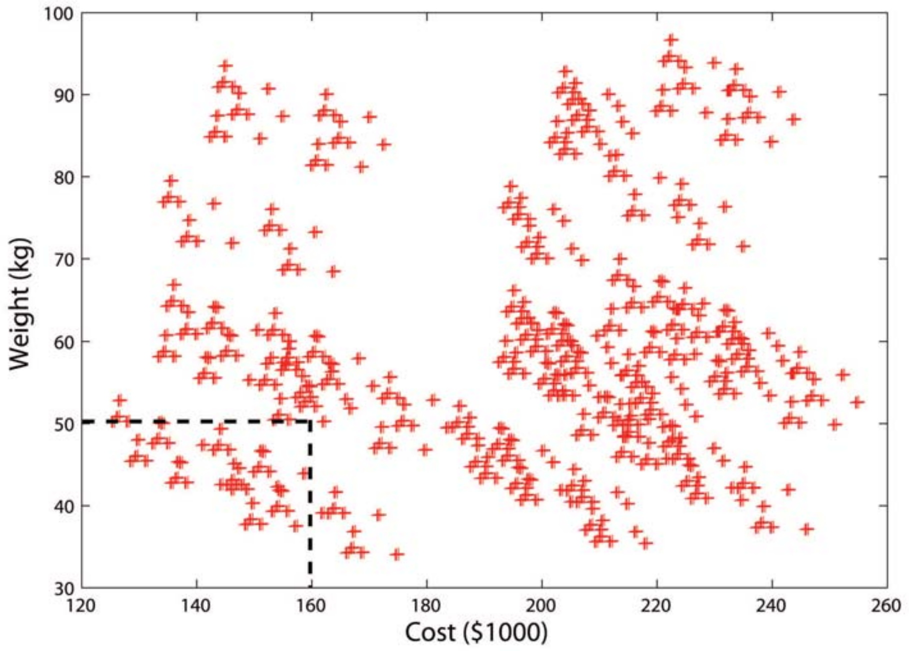


Figure 4.7: Satellite Configurations with Satellite Constraints

Table 4.15: Satellite Configurations Created by Introducing Similar Configuration Elimination Method

Component Level Constraints		yes
Availability Constraints		yes
Connection Type Constraints		yes
Orientation Constraints		yes
Satellite Level Constraints		yes
Similar Configuration Elimination		yes (criteria = 0.08)
Components	**Number of Components in Warehouse**	**Number of available Connection Points**
Sun Sensor	6	4
Earth Sensor	5	1
Thruster	4	4
Payload	4	2
Reaction Wheel	5	3
Number of Satellite Configurations		39

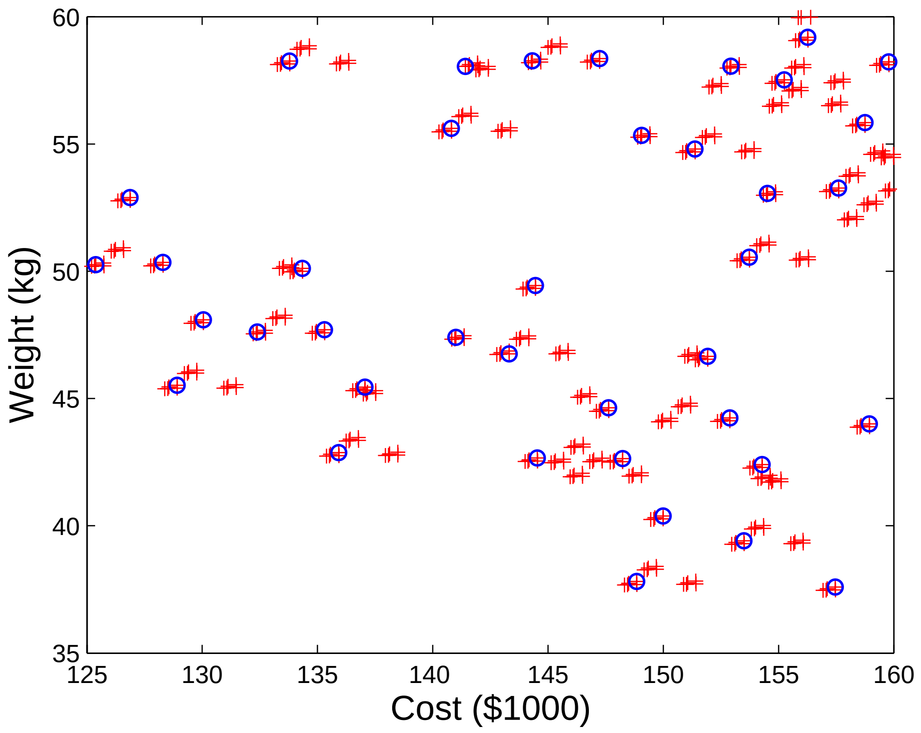


Figure 4.8: Satellite Configurations Eliminated based on Similarity

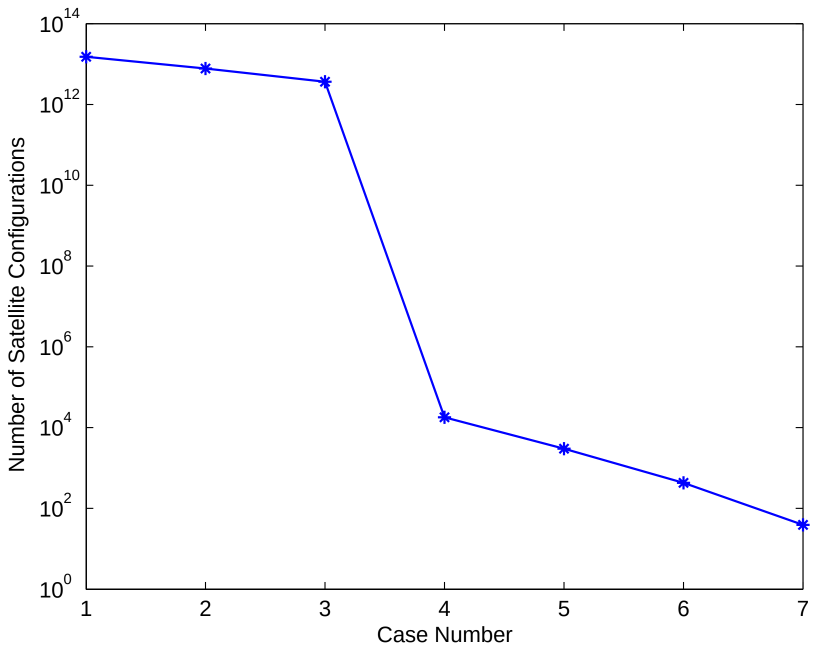


Figure 4.9: Elimination of Satellite Configurations

V. Conclusions and Future Work

Although there are numerous ways tackle the problem of creating the best way to design satellites, we believe that this research effort provides an important contribution in evaluating a "constraint based" approach in configuring PnP satellites. This work has shown that satellite configurations created from PnP components can result in very large numbers of configurations, but they can be reduced to manageable numbers by applying constraints early in the configuration process.

5.1 Conclusions

Most of the approaches explained in Chapter II support the rapid development of only one satellite system by helping developers reduce the time and effort spent on each design phase. In these previously explained approaches, most satellites are created from existing satellite components which are not PnP and the design process requires close iterative user interaction with the software tools. By using a constraint based approach, the "automatic" configuration process of PnP components results in the ability to evaluate a wide variety of satellite configurations quickly.

One key aspect of this approach is reducing the computational costs by excluding components and/or component combinations which don't meet the requirements "as soon as possible" in the process. User defined constraints are applied whenever it is applicable to minimize the effect of "state explosion" -a condition of extraordinarily large numbers of configurations typically caused by a wide variety of components and locations- in creating satellite configurations and wasted computational cycles. Both connection type and orientation attributes of components creates constraints on the number of components that must agree with an associated connection point constraints. Therefore, the magnitude of the "state explosion" can be significantly decreased as more constraints are applied.

Another key aspect of this approach is not allowing using "various" (different brand and model) components that perform the same function. Only allowing "like" (same brand and model) component combinations results in ease in coding in order to handle differences and conflicts between components. Creating PnP component configurations from only "like" PnP components also reduces the complexity and number of satellite configurations. The number of configurations created can easily be in the millions from only a few components and locations which makes it almost impossible to manage due to limited time, computational resources and the user could be overwhelmed trying to determine which configuration to choose. Creating configurations by combining various components is not considered a good methodology to follow because of the exponential increase in the large number of configurations and, more importantly, complexity of handling components with different brands and specifications.

Components and locations of components on the satellite together compose a PnP satellite. In this research, to create combinations of components and their locations we first selected the components and then created component locations with different "orders" for those selected components. Locations with different "order" creates different satellite configurations from same components. Another approach could be created by creating component combinations with different "orders" and then assign those components to selected locations. However, the second approach is not recommended because, there will be thousands of components in the warehouse, and creating component combinations regarding the "order" of components creates extremely large combinations to be evaluated.

Different from conventional satellite design methods, we assumed that all components are prefabricated, tested, and validated beforehand, so that satellite creators don't spend time validating every component/subsytem when they are needed. It's the components and locations of components on the satellite together that really define how PnP satellites are configured. In this research, to create combinations of components and their locations we first select the components and then component locations with different "orders" for those selected components. The locations with different "order" creates different satellite configurations from the same components.

When satellite configurations are created from a fixed set of known components, some satellites may be very similar to each other. It is very likely that some components with similar properties and different stock numbers will exist in the warehouse. The components in the library having very similar specifications that may result in creating "nearly identical" satellite configurations. In this research, each satellite configuration is saved to a database and various configurations are sorted based on cost or weight. An approach of normalizing the attributes of the satellite and eliminating "nearly identical" satellite configurations can significantly reduce the number of satellite configurations that user can select from.

In the end, we created a data structure that accommodates different properties of PnP components, satellite structures, and configured satellites by combining OOD and satellite subsystem hierarchy. Having different attributes at different levels in the system makes coding and calculations easier because it hides the complexity of data structure in different levels. This approach, if applied correctly, should result in a manageable methodology of designing PnP satellites, especially if the bulk of computations are performed beforehand. Overall, if this approach is implemented in a commercial code, the followings features are recommended:

1. Apply constraints as soon as possible during the configuration process, especially if the user knows they want a specific component at a particular location.
2. Implement user friendly GUIs to compliment the data gathering and user adoption.
3. Provide a 3D Viewer to allow users to visualize satellite configurations.
4. Designate location types for components with different functionalities.
5. Present results in a way that allows users to sort satellite configurations based on weight, cost, reliability and space qualification of components.
6. Remove "similar" satellite configurations in the end satellite configurations.

5.2 Recommendations for Future Work

The future integration environment of PnP satellites will be composed of thousands of PnP components and their software components. Current satellite design approaches rely heavily on manual configuration methods of which developers are extremely involved in each step, moreover rapidly configuration of satellites composed of thousands of components is not possible with current approaches. Even from small numbers of components and connection points, a very large number of satellite configurations can be created. Quickly figuring out which components and overall configuration best satisfies the user's constraints, like cost, weight, size, or life span, is very difficult because of complexity of satellite subsystems, numerous of available component choices, limited budged, high reliability requirement of space assets etc. Manufacturing satellites to meet the short time line requirement of ORS requires PnP components to be used. However, satellite developers by themselves are not capable of evaluating all possible configurations in a design space full of PnP components with different attributes and interfaces. This research effort represents the first step for rapidly configuring PnP satellites from PnP components and some suggestions to improve this approaches are given below.

In order to minimize the number of satellite configurations to manageable numbers, various constraints are derived from the user inputs at different levels of the configuration process. User inputs might be very general information such as mission type, mission duration or very specific such as the selection of a particular component that is determined beforehand to be used in the configurations. The more inputs related to mission and PnP satellite, the more constraints can be created and it might be possible to reduce the number of PnP satellite configurations to the manageable numbers. It is very possible to add new constraints at different levels in the PnP satellite configuration process to eliminate more satellite configurations before they are created. The user input list presented in this research is not complete and more inputs can be introduced to this approach. The more detailed inputs given by the user results in more constraints that can be applied which typically result in a significant decrease in the number of satellite configurations.

In this approach, we used “like” components to create configurations. As a future improvement, this restriction could be removed for certain type of components. Using different components can provide flexility to configure satellites, but it also can create some drawbacks like compatibility issues that may not be handled with simple logic. Components made by different manufacturers are likely to have different specifications. Thus, using different components requires more complex coding to handle differences between components to prevent conflicts. As discussed in this research, the number of created satellite configurations can be extremely large in the case of allowing using various components. We think that this approach would be best used to create a database of all satellite configurations beforehand from which various programs evaluate those configurations for user selection.

Typically, satellite systems are complex and estimating performance characteristics of satellite is not a simple process because of the interconnection between the components. In this research, all components are assumed “distinct” meaning the presence of one component has no effect on other components. However, certain components may require other components to be present in a configurations or components meeting a requirement by itself, may not meet the requirements, for example when they are assembled together with other components. On the contrary, there exists sophisticated components that can provide functions that only a group of components can provide together. Using such high tech or multi-function components requires redundant components to be eliminated, otherwise function conflict between the components may be inevitable. In this case, a component dependencies matrix could be created and utilized at both the satellite and subsystem level. It is anticipated that PnP component technologies will offer more standardization in components. An approach is needed to manage dependencies between components in a satellite configuration when the industry standards are well established.

The SMAD approach as discussed in Section 2.2 has methods to estimate of mass, cost, and power budgets etc. of each subsystem or component with respect to the entire satellite based on satellite historical data. Those estimations can be used as constraints to eliminate invalid configurations before satellite configurations are created. However, percentage estimates generated from today's small satellite historical data may lead to poor estimations because PnP Satellites are expected to have different subsystems and component than currently exists. Therefore, a study on PnP components and their attributes such as mass, volume, cost, and reliability based on historical data and industry trend to create an database could be very beneficial.

Flight history of components may be a determining factor for the user in selecting specific components. Satellite configurations may be sorted according to the number or percentage of components that have acceptable records of flight and quite possibly the success in a given configuration to allow user to design the most reliable configuration. This approach can be enhanced by adding an capability that helps users to sort or eliminate satellite configurations based on their components' space qualifications.

In the developed approach, we assumed there is one satellite structure which is selected depending on the mission type and orbit for PnP components to be attached during configuration process. Like PnP components, a variety (small, medium, large) of panels can be used to create various satellite structures. An approach listing a limited variety of satellite structures assembled from those PnP panels could easily be developed to assist the user in selecting the appropriate satellite structure. The selection of satellite structure could also be automated later using a process that takes into account historical data. Mission type, mission duration, orbit, and available booster are other key factors that could be used determining the size of the satellite structure. This approach could also create all possible satellite structures from existing panels and determine the connection points on the panels.

When attaching components to a satellite structure, it is likely that the different size of components could overlap or conflict with each other if we don't consider the dimension of the components. Each satellite structure could have fixed number of specific connection points on the panels of the structure with a fixed distance between connection points, and the distances could be computed from the largest dimensions of different type of component. Each component could have designated connection locations on the satellite structure that does not let components to overlap.

PnP satellites will play a fundamental role in making space more responsive. In the space industry, most satellites are designed to perform space missions which are driven by non-PnP payloads using iterative processes. On the other hand, technologies such as computing power and micro electronics have advanced significantly resulting in increasingly capable payloads that can be fit onto smaller spacecraft. Utilization of PnP technology and an approach like the one presented in this research effort will allow developers to reduce time spent on simple details to free up time for working on the important other challenges to meet the goals of ORS.

Listing A.1: Calculation of Mean and Standard Deviation Method

```
public static double[] calculateMeanandStandartDeviation(double[] ...
    data){
        int n = data.length;
        double ave = 0.0;
        for(int j = 0; j < n; j++){ ave += data[j];}
                ave /= n;
        double variance = 0.0;
        double ep = 0.0;
        double s = 0.0;
        for(int j=0; j<n; j++){
            s = data[j] - ave;
            ep += s;
            variance += s*s;
        }
        variance = (variance - (ep*ep)/n)/(n-1);
        double[] rslt = {ave,Math.sqrt(variance)};
        return rslt;
    }
```

Listing A.2: Generation of Gaussian PDF Values Method

```
public static double createRandomNormalDistNum(double mean, double...
     stdev, int decimal,int division) {

     Random rnd = new Random();
     double number = mean + (stdev/division) * rnd.nextGaussian();
     number = (int) (number * Math.pow(10, decimal) ) / Math.pow...
        (10, decimal);
     return number;
  }
}
```

Listing A.3: Component Class

```
package satkan.persistant;

public class Component {

    public   Component(){};
    public   Component(String Id,String[] region,String ...
       connectionType,double mass,
            double cost,double reliability,int availability,double...
               [] lwh){
        this.Id = Id ;
        this.orientation = region;
        this.connectionType = connectionType;
        this.reliability = reliability;
        this.cost = cost;
        this.mass = mass;
        this.availability = availability;
        this.dimension = lwh;   //lwh
    }

    public String getId() {
        return Id;
    }

    public String[] getRegion() {
        return orientation;
    }

    public String getConnectionType() {
        return connectionType;
    }
    public double getMass() {
        return mass;
    }
    public double getCost() {
        return cost;
    }
   public int getAvailability() {
        return availability;
    }
      public double getReliabiltiy() {
        return reliability;
    }

      public void set_connectionType(String newType) {
        this.connectionType = newType;
    }
       public void set_orientation(String[] newType) {
         this.orientation = newType;
    }
              public void set_availabilit(int availability) {
         this.availability = availability;
    }

public String Id;
public String[] orientation;
public String connectionType;
public double mass;
public double cost;
public double reliability;
public int availability;
public double[] dimension;


}
```

Listing A.4: Attitude and Determination Control Subsystem Class

```
package satkan.persistant;

/**
```

```
 *ADCS Attittude Determination Control System Components
 *Other names:
 *ACS(Attitude Control System)
 *GN&C(Guidancd,Navigation & Control System)
 */
public class ADCS extends Component implements satkan.Momentable {

    public ADCS() {}
    public ADCS(String Id, String[] region, String connectionType,...
        double[] cg, double mass, double[] localInertiaTensor, ...
       double cost, double reliability, int availability, double[]...
        lwh) {
        super(Id, region, connectionType, mass, cost, reliability,...
            availability, lwh);
        this.cg = cg;
        this.localInertiaTensor = localInertiaTensor;
    }
    private double[] cg = new double[3];
    private double[] localInertiaTensor = new double[3];

    public double[] get_cg() {
        return this.cg;
    }

    public double get_mass() {
        return this.mass;
    }

    public double[] get_localInertiaTensor() {
        return this.localInertiaTensor;
    }

    public void set_localInertiaTensor(double[] localInertiaTensor...
       ) {
        this.localInertiaTensor = localInertiaTensor;
    }

    public void set_Id(String Id) {
        this.Id = Id;
    }

    public void set_mass(double mass) {
        this.mass = mass;
    }

    public String get_id() {
        return this.Id;
    }
}
```

Listing A.5: Sun Sensor Class

```
package satkan.persistant;
```

```
public class SunSensor extends AttitudeSensor{

         public SunSensor() {} ;
         public SunSensor(String Id,String[] region,String ...
             connectionType,double[] cg ,
              double mass,double[] localInertiaTensor,double cost,...
                  double reliability,int availability,
              double accuracy,double[] lwh){
     super(Id,region,connectionType,cg,mass,localInertiaTensor,cost...
         ,reliability,availability,accuracy,lwh);
     }
}
```

Listing A.6: Earth Sensor Class

```
package satkan.persistant;

public class EarthSensor extends AttitudeSensor{

         public EarthSensor() {} ;
         public EarthSensor(String Id,String[] region,String ...
             connectionType,double[] cg ,
              double mass,double[] localInertiaTensor,double cost,...
                  double reliability,int availability,
              double accuracy,double[] lwh){
     super(Id,region,connectionType,cg,mass,localInertiaTensor,cost...
         ,reliability, availability, accuracy,lwh);
     }
}
```

Listing A.7: Star Sensor Class

```
package satkan.persistant;

public class StarSensor extends AttitudeSensor {

     public StarSensor() {} ;
     public StarSensor (String Id,String[] region,String ...
         connectionType,double[] cg ,
              double mass,double[] localInertiaTensor,double cost,...
                  double reliability,int availability,
              double accuracy,double[] lwh){
     super(Id,region,connectionType,cg,mass,localInertiaTensor,cost...
         ,reliability,availability,accuracy,lwh);

     }
}
```

Listing A.8: Magnetometer Class

```
package satkan.persistant;

public class Magnetometer extends AttitudeSensor {
    public Magnetometer  (){};

    public Magnetometer (String Id,String[] region,String ...
       connectionType,double[] cg ,
            double mass,double[] localInertiaTensor,double cost,...
               double reliability,int availability,
            double accuracy,double[] lwh){
    super(Id,region,connectionType,cg,mass,localInertiaTensor,cost...
       ,reliability,availability,accuracy,lwh);
    }
}
```

Listing A.9: Gyroscope Class

```
package satkan.persistant;

public class Gyroscope extends AttitudeSensor {

    public Gyroscope() {
    }

    public Gyroscope(String Id, String[] region, String ...
       connectionType, double[] cg, double mass, double[] ...
       localInertiaTensor, double cost, double reliability, int ...
       availability, double accuracy, double[] lwh) {
        super(Id, region, connectionType, cg, mass, ...
           localInertiaTensor, cost, reliability, availability, ...
           accuracy, lwh);
    }
}
```

Listing A.10: GPS Class

```
package satkan.persistant;

public class Gps extends AttitudeSensor {
  public Gps() {} ;
  public Gps(String Id,String[] region,String connectionType,...
     double[] cg ,
       double mass,double[] localInertiaTensor,double cost,double ...
          reliability,int availability, double accuracy,double[] ...
          lwh){
    super(Id,region,connectionType,cg,mass,localInertiaTensor,cost...
       ,reliability,availability,accuracy,lwh);

    }
}
```

Listing A.11: Inertial Sensor Class

```
package satkan.persistant;

public class InertialSensor extends AttitudeSensor {
    public InertialSensor() {
    }

    public InertialSensor(String Id, String[] region, String ...
       connectionType, double[] cg, double mass, double[] ...
       localInertiaTensor, double cost, double reliability, int ...
       availability, double accuracy, double[] lwh) {
        super(Id, region, connectionType, cg, mass, ...
           localInertiaTensor, cost, reliability, availability, ...
           accuracy, lwh);
    }
}
```

Listing A.12: Reaction Wheel Class

```
package satkan.persistant;

public class ReactionWheel  extends AttitudeActuator{
    public ReactionWheel() {} ;
        public ReactionWheel(String Id,String[] region,String ...
           connectionType,double[] cg ,
            double mass,double[] localInertiaTensor,double cost,...
               double reliability,int availability,
            double torque,double[] lwh){
    super(Id,region,connectionType,cg,mass,localInertiaTensor,cost...
       ,reliability,availability,torque,lwh);
    }
}
```

Listing A.13: Thruster Class

```
package satkan.persistant;

public class Thruster extends AttitudeActuator {
      public Thruster() {} ;
        public Thruster(String Id,String[] region,String ...
           connectionType,double[] cg ,
            double mass,double[] localInertiaTensor,double cost,...
               double reliability,int availability,
            double torque,double[] lwh){
    super(Id,region,connectionType,cg,mass,localInertiaTensor,cost...
       ,reliability,availability,torque,lwh);
    }
}
```

Listing A.14: Satellite Class

```
package satkan.persistant;

import satkan.*;
import java.util.ArrayList;

public class Satellite implements Momentable {
    public Satellite(){}

    public Satellite(String Id, ArrayList<Momentable> list, ...
       ArrayList<ComponentLocation> location) {
        this.Id = Id;
        this.list = list;
        this.location = location;
        this.mass = this.get_mass();
        this.cost = this.get_cost();
        this.reliability = this.get_reliability();
    }
    public String Id;
    public ArrayList<Momentable> list;
    public ArrayList<ComponentLocation> location;
    public double cost;
    public double mass;
    public double reliability;

    public ArrayList get_component_list() {
        return list;
    }

    public double[] get_cg() {
        throw new UnsupportedOperationException("Not supported yet...
            .");
    }

    public double get_reliability() {
        reliability = 0.0;
        for (int lis = 0; lis < list.size(); lis++) {
            Component comp = (Component) list.get(lis);
            reliability = reliability + comp.getReliabiltiy();
        }
        return reliability / list.size();
    }

    public double get_mass() {
        mass = 0;
        for (int lis = 0; lis < list.size(); lis++) {
            Component comp = (Component) list.get(lis);
            mass = mass + comp.getMass();
        }
        return mass;
    }

    public double get_cost() {
        cost = 0;
        for (int lis = 0; lis < list.size(); lis++) {
            Component comp = (Component) list.get(lis);
            cost = cost + comp.getCost();
        }
        return cost;
    }

    public double[] get_localInertiaTensor() {
        throw new UnsupportedOperationException("Not supported yet...
            .");
    }

    public String get_id() {
        return this.Id;
    }

    public ArrayList<Momentable> get_satellite_components() {
        return this.list;
    }

    public ArrayList<double[]> get_satellite_rotation_local() {
        ArrayList<double[]> rotation_local = new ArrayList<double...
            []>();
        for (int i = 0; i < this.location.size(); i++) {
            rotation_local.add(this.location.get(i)....
                getLocalRotation());
        }
        return rotation_local;
    }

    public ArrayList<double[]> get_satellite_rotation_global() {
        ArrayList<double[]> rotation_global = new ArrayList<double...
            []>();
        for (int i = 0; i < this.location.size(); i++) {
            rotation_global.add(this.location.get(i)....
                getGlobalRotation());
        }
        return rotation_global;
    }

    public ArrayList<double[]> get_satellite_translation() {
        ArrayList<double[]> translation = new ArrayList<double...
            []>();
        for (int i = 0; i < this.location.size(); i++) {
            translation.add(this.location.get(i).getTranslation())...
                ;
        }
        return translation;
    }
}
```

Listing A.15: Plate Class

```
package satkan.persistant;

import javax.vecmath.AxisAngle4f;
import satkan.*;
public class Plate extends Structure  {
   public Plate (){};
   public Plate(String Id,String[] region,String connectionType,
      double[] xyz,double density,
      double cost,double reliability,int availability,double[] lwh...
         ){
      super(Id,region,connectionType,xyz,0,null,cost,reliability, ...
         availability,lwh);
        this.height = lwh[2];
        this.width = lwh[1];
        this.thickness = lwh[0];
        this.density = density ;
        this.volume = height * width * thickness ;
        double mass1 = density * this.volume;
      double[] localInertiaTensor = new double[3] ;
      localInertiaTensor[0] = 1.0/12.0 * mass1 *height * height ;
      localInertiaTensor[1] = 1.0/12.0 * mass1 * width *width ;
      localInertiaTensor[2] = 1.0/12.0 * mass1 * ( height * height...
          + width * width);
      super.set_localInertiaTensor(localInertiaTensor);
      super.set_mass(mass1);
                }
private double height;
private double width;
private double thickness;
private double density;
private double volume ;

    public double  get_height()
        {
        return height ;
        }
    public double  get_width()
        {
        return width ;
        }
    public double  get_thickness()
       {
        return thickness ;
}
        public double  get_density()
        {
        return density ;
}
    public double  get_volume()
        {
        return volume ;
        }
 public void  set_volume(double volume)
        {
        this.volume = volume;
        }
}
```

Listing A.16: Create Object Combination List Method

```
/**  * Creates combinations of component ids using given ...
   parameters and returns them in ArrayList<Object[]>
     * @param components Ids of the same type of components in ...
        Arrays
     * @param numberofUsage  Number of components in a single ...
        combination to be created
     * @param repetitionOnly true : creates list of combinations ...
        using only  'like' component Id
     *                       false: creates list of combinations ...
        using depends on the "repetition" parameter
     * @param repetition  true: creates list of combinations using...
          the same component Ids and other component Ids as well. ...
        Works only  when "repetitiononly" is "false"
     *                    false: creates list of combinations ...
        using only different components Ids.  Works only  when "...
        repetitiononly" is "false"
     *
     * @return contains Arrays of  component combinations
     * @exception no exception
     */
    public static ArrayList<Object[]> createObjectCombinationList(...
       Object[] components, int numberofUsage, boolean same, ...
       boolean different) throws CombinatoricException {
        ArrayList<Object[]> combinedList = new ArrayList<Object...
           []>();
        // Only one type of component combination will be added
        if (same) {
            for (int componentsIndex = 0; componentsIndex < ...
               components.length; componentsIndex++) {
                Object[] combo = new Object[numberofUsage];
                for (int comboIndex = 0; comboIndex < ...
                   numberofUsage; comboIndex++) {
                    combo[comboIndex] = components[componentsIndex...
                       ];
                }
                combinedList.add(combo);
            }
        } else {
            Combinations c = new Combinations(components, ...
               numberofUsage);
            while (c.hasMoreElements()) {
                Object[] combo = (Object[]) c.nextElement();
                combinedList.add(combo);
            }
        } //add same type of component combination
        if (different && (numberofUsage > 1) && (!same)) {
            for (int componentsIndex = 0; componentsIndex < ...
               components.length; componentsIndex++) {
                Object[] combo = new Object[numberofUsage];
                for (int comboIndex = 0; comboIndex < ...
                   numberofUsage; comboIndex++) {
                        combo[comboIndex] = components[componentsIndex...
                            ];
                    }
                    combinedList.add(combo);
                }

        } else if (combinedList.get(0).length == 0) {
        }
        return combinedList;
    }
```

Listing A.17:   Create Combined Object Combination List Method

```
    /**  * Creates a new combined  ArrayList<Object[]> from ...
        different ArrayList<Objects[]>
     *
     * @param list ArrayList contains  ArrayList<Objects[]> which ...
        contains  combinations of component Ids.
     * @return  ArrayList<Objects[]>
     * @exception no exception
     */
    public static ArrayList<Object[]> ...
        createCombinedObjectCombinationList(ArrayList<ArrayList<...
        Object[]>> list) {

        ArrayList<Object[]> combinedList = new ArrayList<Object...
            []>();
        ArrayList<Object[]> tempCombinedList = new ArrayList<...
            Object[]>();
        ArrayList<Object[]> ArrayListTemp = new ArrayList<Object...
            []>();
        for (int i = 0; i < list.size(); i++) {
            ArrayListTemp = list.get(i);
            if (combinedList.isEmpty()) {
                combinedList = ArrayListTemp;
            } else {
                int t = 0;
                int t2 = 0;
                for (int tempIndex = 0; tempIndex < ArrayListTemp....
                    size(); tempIndex++) {
                    Object[] ArrayListTempRow = ArrayListTemp.get(...
                        tempIndex);
                    t2++;
                    for (int combinedIndex = 0; combinedIndex < ...
                        combinedList.size(); combinedIndex++) {

                        Object[] combinedRow = combinedList.get(...
                            combinedIndex);
                        t++;
                        Object[] newCombinedRow = new Object[...
                            combinedRow.length + ArrayListTempRow....
                            length];
                        for (int j = 0; j < combinedRow.length; j...
                            ++) {
                             newCombinedRow[j] = combinedRow[j];
                        }

                        for (int tempRowIndex = 0; tempRowIndex < ...
                            ArrayListTempRow.length; tempRowIndex...
                            ++) {
                             newCombinedRow[combinedRow.length + ...
                                 tempRowIndex] = ArrayListTempRow[...
                                 tempRowIndex];
                        }

                        tempCombinedList.add(newCombinedRow);
                    }
                }
                combinedList.clear();
                combinedList.addAll(tempCombinedList);
                tempCombinedList.clear();
            }
        }
        System.out.println(combinedList.size());
        tempCombinedList = null;
        ArrayListTemp = null;
        list = null;
        return combinedList;
    }
```

Listing A.18: Remove Under Constraint C3s Method

```
public static void removeUnderConstraintC3s(ArrayList<Object[]> ...
    C3L, double c_mass, double c_cost) throws ...
    ClassNotFoundException {
        DataBase db = new DataBase();
        for (int C3LRowIndex = 0; C3LRowIndex < C3L.size(); ...
            C3LRowIndex++) {
             Object[] C3LRow = C3L.get(C3LRowIndex);
             //create mass variable
             double mass = 0;
             double cost = 0;
             for (int i = 0; i < C3LRow.length; i++) {
                  String Id = (String) C3LRow[i];
                  Component comp = (Component) db....
                      get_component_object("Component", Id);
                  mass = mass + comp.getMass();
                  cost = cost + comp.getCost();
             }
             if (mass > c_mass) {
                  C3L.remove(C3LRowIndex);
                  C3LRowIndex--;
             } else if (cost > c_cost) {
                  C3L.remove(C3LRowIndex);
                  C3LRowIndex--;
            }
        }
    }
```

Listing A.19: Remove Invalid C3s Method

```
public static void removeInvalidC3s(ArrayList<Object[]> C3L) ...
    throws ClassNotFoundException {
        DataBase db = new DataBase();
        ArrayList<String[]> typeCount = new ArrayList<String[]>();
        typeCount = db.get_ConnectionTypes();
        for (int C3LRowIndex = 0; C3LRowIndex < C3L.size(); ...
            C3LRowIndex++) {
            Object[] C3LRow = C3L.get(C3LRowIndex);
            String[] conTypes = new String[C3LRow.length];
            //get all connection types corresponding the components...
                 Ids
            for (int i = 0; i < C3LRow.length; i++) {

                String Id = (String) C3LRow[i];
                Component comp = (Component) db....
                    get_component_object("Component", Id);
                conTypes[i] = comp.getConnectionType();
            }
            String[] typeCountSat;
            typeCountSat = Assemble.countSameObj(conTypes);
            for (int i = 0; i < typeCountSat.length; i++) {
                boolean hasType = false;
                for (String[] temp : typeCount) {
                    if (temp[0].equals(typeCountSat[i])) {
                        hasType = true;
                        if (Double.parseDouble(temp[1]) < Double....
                            parseDouble(typeCountSat[i + 1])) {
                             C3L.remove(C3LRowIndex);
                             C3LRowIndex--;
                        }
                    }
                }
                if (!hasType) {
                    C3L.remove(C3LRowIndex);
                    C3LRowIndex--;
                } else {
                }
                i++;
            }
        }
    }
```

Listing A.20: Create Combined Object Combination Location List Method

```
public static ArrayList<ArrayList<Object[]>> ...
    createCombinedObjectCombinationLocationListList(ArrayList<...
    Object[]> C3L) throws ClassNotFoundException, ...
    CombinatoricException {
```

```
    //open the database
    DataBase db = new DataBase();
    //define the list to be returned
    ArrayList<ArrayList<Object[]>> locationList = new ...
       ArrayList<ArrayList<Object[]>>();
    //use  every row in the C3L (has only components)
    for (int listRow = 0; listRow < C3L.size(); listRow++) {
        //define  an arraylist
        ArrayList<ArrayList<Object[]>> RowLocations = new ...
           ArrayList<ArrayList<Object[]>>(); //
        //get the components into Array
        Object[] C3LRow = C3L.get(listRow);
        // find the repetition of each component to create ...
           combination
        String[] ComponentCount;
        ComponentCount = Assemble.countSameObj(C3LRow);
        //remove repeated Components
        ArrayList<Object> temparray = new ArrayList<Object>();
        Set s = new HashSet();
        for (int i = 0; i < C3LRow.length; i++) {
            if (s.add(C3LRow[i])) {
                temparray.add(C3LRow[i]);
            }
        }
        C3LRow = temparray.toArray();
        // for every component in the combinedRow  cast Id ...
           object to String so that it can be used to reach ...
           the db
        for (int row = 0; row < C3LRow.length; row++) {
            String Id = (String) C3LRow[row];
            Component comp = (Component) db....
               get_component_object("Component", Id); //get ...
               real object from db
            String conType = comp.getConnectionType();
            String[] region = comp.getRegion();
            ArrayList<Object[]> componentLocations = new ...
               ArrayList<Object[]>(); // all locations for one...
                component (region + connectionType)
            //there may be more than one region the component ...
               may be used
            for (int reg = 0; reg < region.length; reg++) {
                //get location Ids  of which region and ...
                   connection type match
                String[] str = db.get_Id("ComponentLocation", ...
                   "region", region[reg], "connectionType", ...
                   conType);
                for (int strCount = 0; strCount < str.length; ...
                   strCount++) {
                    Object[] tempObj = new Object[1];
                    if (!str[strCount].isEmpty()) {
                        tempObj[0] = str[strCount];
                        componentLocations.add(tempObj);
                    }
                }
            }

            if (!(componentLocations.get(0).length < 1)) {
                Object[] CLArray = new Object[...
                   componentLocations.size()];
                for (int row2 = 0; row2 < componentLocations....
                   size(); row2++) {
                    // In array List its in Object[] mode, one...
                         level upper needed
                    CLArray[row2] = componentLocations.get(...
                       row2)[0];
                }
                String temp = ComponentCount[row * 2 + 1];
                int numberofusage = Integer.parseInt(temp);
                componentLocations = Assemble....
                   createObjectCombinationList(CLArray, ...
                   numberofusage, false, false);
                RowLocations.add(componentLocations);
            } else {
                // dont need to check the other components ...
                   since it is not a valid C3
                break;
            }
        }

        ArrayList<Object[]> rowLocationList = new ArrayList<...
           Object[]>();
        if (C3LRow.length == RowLocations.size()) {
            rowLocationList = Assemble....
               createCombinedObjectCombinationList(...
               RowLocations);
            //Eliminate Location Rows with repeated locations
            for (int k = 0; k < rowLocationList.size(); k++) {
                Object[] temprow = rowLocationList.get(k);
                String[] tempnum = Assemble.countSameObj(...
                   temprow);
                for (int temp = 0; temp < tempnum.length; temp...
                   ++) {
                    temp++;
                    if (!(tempnum[temp].equals("1"))) {
                        rowLocationList.remove(k);
                        k--;
                        break;
                    }
                }
            }

            locationList.add(rowLocationList);
        } else {
            C3L.remove(listRow);
```

```
                    listRow--;
                }
            }
            return locationList;
        }
    }
```

Listing A.21: Construct Viewer Class

```
package satkan;

import math.*;
import com.sun.j3d.loaders.Scene;
import com.sun.j3d.loaders.objectfile.ObjectFile;
import javax.swing.JFrame;
import java.awt.*;
import javax.swing.*;
import javax.media.j3d.Canvas3D;
import com.sun.j3d.utils.universe.SimpleUniverse;
import javax.media.j3d.BranchGroup;
import com.sun.j3d.utils.geometry.Box;
import javax.vecmath.*;
import javax.media.j3d.DirectionalLight;
import javax.media.j3d.BoundingSphere;
import javax.media.j3d.Appearance;
import javax.media.j3d.Material;
import javax.media.j3d.TransformGroup;
import com.sun.j3d.utils.behaviors.mouse.*;
import com.sun.j3d.utils.geometry.Cone;
import com.sun.j3d.utils.geometry.Cylinder;
import com.sun.j3d.utils.geometry.Sphere;
import com.sun.j3d.utils.universe.ViewingPlatform;
import javax.media.j3d.ColoringAttributes;
import javax.media.j3d.Font3D;
import javax.media.j3d.FontExtrusion;
import javax.media.j3d.Shape3D;
import javax.media.j3d.Text3D;
import javax.media.j3d.Transform3D;

public class ConstructViewer extends JFrame {
  /**
   * The SimpleUniverse object
   */
  protected SimpleUniverse simpleU;

  /**
   * The root BranchGroup Object.
   */
  protected BranchGroup rootBranchGroup;

  /**
   * Constructor that consturcts the window with the given name.
   *
   * @param name
   *            The name of the window, in String format
   */
  public ConstructViewer(String name) {
    // The next line will construct the window and name it
    // with the given name
    super(name);

    // Perform the initial setup, just once
    initial_setup();
  }

  /**
   * Perform the essential setups for the Java3D
   */
  protected void initial_setup() {

    getContentPane().setLayout(new BorderLayout());
    GraphicsConfiguration config = SimpleUniverse
        .getPreferredConfiguration();

    Canvas3D canvas3D = new Canvas3D(config);


    getContentPane().add("Center", canvas3D);
    simpleU = new SimpleUniverse(canvas3D);
    simpleU.getViewingPlatform().setNominalViewingTransform();
    rootBranchGroup = new BranchGroup();
  }
  public void addDirectionalLight(Point3d boundingPoint, double ...
     boundingRadius ,Vector3f direction, Color3f color) {
    // Creates a bounding sphere for the lights
    BoundingSphere bounds = new BoundingSphere(boundingPoint,...
       boundingRadius);
    // Then create a directional light with the given
    DirectionalLight lightD = new DirectionalLight(color, ...
       direction);
    lightD.setInfluencingBounds(bounds);
    // Then add it to the root BranchGroup
    rootBranchGroup.addChild(lightD);
  }
  public void addPart(String name, String colorStr, Color3f spec,
          Vector3f transVector, Matrix3d  Rotation){
    TransformGroup tg = new TransformGroup();
    Appearance app = new Appearance();
    Material mat = new Material();
    mat.setSpecularColor(spec);
    mat.setShininess(5.0f);
    app.setMaterial(mat);

    ObjectFile file = new ObjectFile();
    Scene scene = null;
    try {
        scene =  file.load(ClassLoader.getSystemResource(name+"....
           obj"));

    } catch (Exception e) {
    }

    Transform3D translate = new Transform3D();

    translate.set(transVector);


    Transform3D rotate = new Transform3D();
    rotate.set(Rotation);
    TransformGroup plateTGR = new TransformGroup(rotate);
    System.out.println(rotate.getScale());
    plateTGR.addChild(scene.getSceneGroup());
    TransformGroup plateTGT = new TransformGroup(translate);
    tg.addChild(plateTGT);

    plateTGT.addChild(plateTGR);

    // Then add it to the rootBranchGroup
    rootBranchGroup.addChild(tg);

    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

    MouseRotate myMouseRotate = new MouseRotate();
    myMouseRotate.setTransformGroup(tg);
    myMouseRotate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseRotate);

    MouseTranslate myMouseTranslate = new MouseTranslate();
    myMouseTranslate.setTransformGroup(tg);
    myMouseTranslate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseTranslate);

    MouseZoom myMouseZoom = new MouseZoom();
    myMouseZoom.setTransformGroup(tg);
    myMouseZoom.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseZoom);
    }

  public void addSphere(float x, Color3f diffuse, Color3f spec,
           Vector3f transVector, AxisAngle4f  axisRotation){
    TransformGroup tg = new TransformGroup();
    // First setup an appearance for the obj
    Appearance app = new Appearance();
    Material mat = new Material();
    mat.setDiffuseColor(diffuse);
    mat.setSpecularColor(spec);
    mat.setShininess(5.0f);
    app.setMaterial(mat);
    Sphere obj = new Sphere(x);
    obj.setAppearance(app);

    Transform3D translate = new Transform3D();
    translate.set(transVector);
    TransformGroup plateTGT = new TransformGroup(translate);
    tg.addChild(plateTGT);

    Transform3D rotate = new Transform3D();

    Matrix3d matrix = new Matrix3d();
    matrix.set(axisRotation);
    rotate.set(matrix);
    TransformGroup plateTGR = new TransformGroup(rotate);

    plateTGR.addChild(obj);
    plateTGT.addChild(plateTGR);

    // Then add it to the rootBranchGroup
    rootBranchGroup.addChild(tg);

    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

    MouseRotate myMouseRotate = new MouseRotate();
    myMouseRotate.setTransformGroup(tg);
    myMouseRotate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseRotate);

    MouseTranslate myMouseTranslate = new MouseTranslate();
    myMouseTranslate.setTransformGroup(tg);
    myMouseTranslate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseTranslate);

    MouseZoom myMouseZoom = new MouseZoom();
    myMouseZoom.setTransformGroup(tg);
    myMouseZoom.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseZoom);
    }
    public void addCylinder(float x, float y, Color3f diffuse, ...
       Color3f spec,
           Vector3f transVector, AxisAngle4f  axisRotation){
    TransformGroup tg = new TransformGroup();

    // First setup an appearance for the obj
    Appearance app = new Appearance();
    Material mat = new Material();
    mat.setDiffuseColor(diffuse);
    mat.setSpecularColor(spec);
    mat.setShininess(5.0f);
    app.setMaterial(mat);
    Cylinder obj = new Cylinder(x, y);
    obj.setAppearance(app);




    Transform3D translate = new Transform3D();
    translate.set(transVector);
    TransformGroup plateTGT = new TransformGroup(translate);
    tg.addChild(plateTGT);

    Transform3D rotate = new Transform3D();

    Matrix3d matrix = new Matrix3d();
    matrix.set(axisRotation);
    rotate.set(matrix);
    TransformGroup plateTGR = new TransformGroup(rotate);

    plateTGR.addChild(obj);
    plateTGT.addChild(plateTGR);

    // Then add it to the rootBranchGroup
    rootBranchGroup.addChild(tg);

    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

    MouseRotate myMouseRotate = new MouseRotate();
    myMouseRotate.setTransformGroup(tg);
    myMouseRotate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseRotate);

    MouseTranslate myMouseTranslate = new MouseTranslate();
    myMouseTranslate.setTransformGroup(tg);
    myMouseTranslate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseTranslate);

    MouseZoom myMouseZoom = new MouseZoom();
    myMouseZoom.setTransformGroup(tg);
    myMouseZoom.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseZoom);
    }
  public void addCone(float x, float y, Color3f diffuse, Color3f ...
     spec,
          Vector3f transVector, AxisAngle4f  axisRotation){
    TransformGroup tg = new TransformGroup();

    // First setup an appearance for the obj
    Appearance app = new Appearance();
    Material mat = new Material();
    mat.setDiffuseColor(diffuse);
    mat.setSpecularColor(spec);
    mat.setShininess(5.0f);
    app.setMaterial(mat);
    Cone obj = new Cone(x, y);
    obj.setAppearance(app);

    Transform3D translate = new Transform3D();
    translate.set(transVector);
    TransformGroup plateTGT = new TransformGroup(translate);
    tg.addChild(plateTGT);

    Transform3D rotate = new Transform3D();

    Matrix3d matrix = new Matrix3d();
    matrix.set(axisRotation);
    rotate.set(matrix);
    TransformGroup plateTGR = new TransformGroup(rotate);

    plateTGR.addChild(obj);
    plateTGT.addChild(plateTGR);

    // Then add it to the rootBranchGroup
    rootBranchGroup.addChild(tg);

    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

    MouseRotate myMouseRotate = new MouseRotate();
    myMouseRotate.setTransformGroup(tg);
    myMouseRotate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseRotate);

    MouseTranslate myMouseTranslate = new MouseTranslate();
    myMouseTranslate.setTransformGroup(tg);
    myMouseTranslate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseTranslate);

    MouseZoom myMouseZoom = new MouseZoom();
    myMouseZoom.setTransformGroup(tg);
    myMouseZoom.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseZoom);

  }
  public void addBox(float x, float y, float z, Color3f diffuse, ...
      Color3f spec,
             Vector3f transVector, AxisAngle4f  axisRotation) {
   TransformGroup tg = new TransformGroup();

    // First setup an appearance for the obj
    Appearance app = new Appearance();
    Material mat = new Material();
    mat.setDiffuseColor(diffuse);
    mat.setSpecularColor(spec);
    mat.setShininess(5.0f);
    app.setMaterial(mat);
    Box obj = new Box(x, y, z, app);

    Transform3D translate = new Transform3D();
    translate.set(transVector);
    TransformGroup plateTGT = new TransformGroup(translate);
    tg.addChild(plateTGT);

    Transform3D rotate = new Transform3D();

    Matrix3d matrix = new Matrix3d();
    matrix.set(axisRotation);
    rotate.set(matrix);
    TransformGroup plateTGR = new TransformGroup(rotate);

    plateTGR.addChild(obj);
    plateTGT.addChild(plateTGR);

    // Then add it to the rootBranchGroup
    rootBranchGroup.addChild(tg);

    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

    MouseRotate myMouseRotate = new MouseRotate();
    myMouseRotate.setTransformGroup(tg);
    myMouseRotate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseRotate);

    MouseTranslate myMouseTranslate = new MouseTranslate();
    myMouseTranslate.setTransformGroup(tg);
    myMouseTranslate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseTranslate);

    MouseZoom myMouseZoom = new MouseZoom();
    myMouseZoom.setTransformGroup(tg);
    myMouseZoom.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseZoom);
  }
public void addArrow( Color3f diffuse, Color3f spec,
          Vector3f transVector, AxisAngle4f  axisRotation) {
   TransformGroup tg = new TransformGroup();

    Axis obj = new Axis();


    Transform3D translate = new Transform3D();
    translate.set(transVector);
    TransformGroup plateTGT = new TransformGroup(translate);
    tg.addChild(plateTGT);

    Transform3D rotate = new Transform3D();

    Matrix3d matrix = new Matrix3d();
    matrix.set(axisRotation);
    rotate.set(matrix);
    TransformGroup plateTGR = new TransformGroup(rotate);

    plateTGR.addChild(obj);
    plateTGT.addChild(plateTGR);

    // Then add it to the rootBranchGroup
    rootBranchGroup.addChild(tg);

    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

    MouseRotate myMouseRotate = new MouseRotate();
    myMouseRotate.setTransformGroup(tg);
    myMouseRotate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseRotate);

    MouseTranslate myMouseTranslate = new MouseTranslate();
    myMouseTranslate.setTransformGroup(tg);
    myMouseTranslate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseTranslate);

    MouseZoom myMouseZoom = new MouseZoom();
    myMouseZoom.setTransformGroup(tg);
    myMouseZoom.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseZoom);
  }

public void  add3dFont(String text, Color3f color,
          Vector3f transVector, AxisAngle4f  axisRotation) {
   TransformGroup tg = new TransformGroup();
     Appearance textAppear = new Appearance();
    ColoringAttributes textColor = new ColoringAttributes();
    textColor.setColor(color);
    textAppear.setColoringAttributes(textColor);
    textAppear.setMaterial(new Material());
    Font3D font3D = new Font3D(new Font("Helvetica", Font.PLAIN, ...
       1),
    new FontExtrusion());
    Text3D textGeom = new Text3D(font3D, text);
    textGeom.setAlignment(Text3D.ALIGN_CENTER);
    Shape3D textShape = new Shape3D();
    textShape.setGeometry(textGeom);
    textShape.setAppearance(textAppear);
    Transform3D translate = new Transform3D();
    translate.set(transVector);
    translate.setScale(0.02);
    TransformGroup plateTGT = new TransformGroup(translate);
    tg.addChild(plateTGT);

    Transform3D rotate = new Transform3D();

    Matrix3d matrix = new Matrix3d();
    matrix.set(axisRotation);
    rotate.set(matrix);
    TransformGroup plateTGR = new TransformGroup(rotate);

    plateTGR.addChild(textShape);
    plateTGT.addChild(plateTGR);

    // Then add it to the rootBranchGroup
    rootBranchGroup.addChild(tg);

    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

    MouseRotate myMouseRotate = new MouseRotate();
    myMouseRotate.setTransformGroup(tg);
    myMouseRotate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseRotate);

    MouseTranslate myMouseTranslate = new MouseTranslate();
    myMouseTranslate.setTransformGroup(tg);
    myMouseTranslate.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseTranslate);

    MouseZoom myMouseZoom = new MouseZoom();
    myMouseZoom.setTransformGroup(tg);
    myMouseZoom.setSchedulingBounds(new BoundingSphere());
    rootBranchGroup.addChild(myMouseZoom);
  }

     public static double[] get_translation(int listIndex){


        double[] arrayTr = SatView.listTr.get(listIndex);
        return arrayTr;
    }
     public static double[] get_rotation_local(int listIndex){

        double[] arrayRt = SatView.listRtl.get(listIndex);
        return arrayRt;
    }

  /**
   * Finalise everything to get ready
```

```
 */
public void finalise() {
  //  add the branch group into the Universe
  simpleU.addBranchGraph(rootBranchGroup);

  // And set up the camera position
  //First get the ViewPlatform:
  ViewingPlatform vp = simpleU.getViewingPlatform();

  //Next get the TransformGroup attached to that ViewPlatform:
  TransformGroup View_TransformGroup = vp.getMultiTransformGroup...
     ().getTransformGroup(0);

  // manipulate the ViewPlatform  create a Transform3D so that ...
     we can move the TransformGroup.
  Transform3D View_Transform3D = new Transform3D();
  View_TransformGroup.getTransform(View_Transform3D);
  // set the translation and assign it to its TransformGroup
  View_Transform3D.setTranslation(new Vector3f(0.0f,0.0f,3.0f));
  View_TransformGroup.setTransform(View_Transform3D);
}

  @SuppressWarnings("deprecation")
public static void main(String[] args) {
 ConstructViewer bc = new ConstructViewer("SATELLITE BUILDER ...
    1.0.0");
 bc.setSize(1200, 1200);
 bc.setTitle("Satellite Viewer");
 bc.setResizable(true);

 for (int i = 0 ;i < SatView.listObj.size();i++){
     String name = SatView.listObj.get(i).get_id();
     String colorStr = "yellow";
     Color3f spec =new Color3f(1, 0, 0);

     Vector3f transVector = new Vector3f() ;
     double[] vec ;
     vec = ConstructViewer.get_translation(i);
     transVector.setX((float)vec[0]) ;
     transVector.setY((float)vec[1]) ;
     transVector.setZ((float)vec[2]) ;

     Matrix3d rotation = new Matrix3d();
     double[] rot ;
     rot = ConstructViewer.get_rotation_local(i);
     rotation.set(rot);

     bc.addPart(name, colorStr, spec, transVector, rotation);
 }

  bc.addCylinder(0.003f, 1.6f, new Color3f(0, 0, 1), new Color3f...
     (0, 0, 1),
            new Vector3f(0.0f, 0.0f,0.0f),new AxisAngle4f( 1.0f,0.0...
                f,0.0f,(float)(90.0/180.0 *Math.PI)));
    bc.addCone(0.006f, 0.017f, new Color3f(1,1,1), new Color3f(1, ...
        1, 1),
            new Vector3f(0f, 0f,0.8f),new AxisAngle4f( 1.0f,0.0f...
                ,0.0f,(float)(90.0/180.0 *Math.PI)));
    bc.add3dFont("+z", new Color3f(1,1,0), new Vector3f(0f, 0f...
        ,0.81f), new AxisAngle4f( 1.0f,0.0f,0.0f,(float)(0.0/180.0 ...
        *Math.PI)));
    bc.addCylinder(0.003f, 1.2f, new Color3f(0,1, 0), new Color3f...
        (0, 1, 0),
            new Vector3f(0.0f, 0.0f,0.0f),new AxisAngle4f( 1.0f,0.0...
                f,0.0f,(float)(0.0/180.0 *Math.PI)));
    bc.addCone(0.006f, 0.017f, new Color3f(1,1, 1), new Color3f(1,...
         1, 1),
            new Vector3f(0f, 0.6f,0f),new AxisAngle4f( 1.0f,0.0f...
                ,0.0f,(float)(0/180.0 *Math.PI)));
    bc.add3dFont("+y", new Color3f(1,1,0), new Vector3f(0f, 0.61f...
        ,0f), new AxisAngle4f( 1.0f,0.0f,0.0f,(float)(-90.0/180.0 *...
        Math.PI)));
    bc.addCylinder(0.003f, 1.6f, new Color3f(1,0, 0), new Color3f...
        (1, 0, 0),
            new Vector3f(0.0f, 0.0f,0.0f),new AxisAngle4f( 0.0f,0.0...
                f,1.0f,(float)(90/180.0 *Math.PI)));
    bc.addCone(0.006f, 0.017f, new Color3f(1,1, 1), new Color3f(1,...
         1, 1),
            new Vector3f(0.8f, 0f,0f),new AxisAngle4f( 0.0f,0.0f...
                ,1.0f,(float)(-90.0/180.0 *Math.PI)));
    bc.add3dFont("+x", new Color3f(1,1,0), new Vector3f(0.81f, 0.0...
        f,0f), new AxisAngle4f( 0.0f,1f,0.0f,(float)(90.0/180.0 *...
        Math.PI)));
    bc.addDirectionalLight(new Point3d (0.0, 0.0,-1.0),300,new ...
        Vector3f(0.0f, 0.0f,-1.0f), new Color3f(1f, 1f, 1f));
    bc.addDirectionalLight(new Point3d (-40, 60,60),150,new ...
        Vector3f(1f, 0f,1f), new Color3f(1f, 1f, 1f));
    bc.finalise();
    bc.setVisible(true);
    return;
  }
}
```

Listing A.22: Database Class

```
package satkan;

import satkan.persistant.Satellite;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import javax.jdo.PersistenceManager;
import javax.jdo.Query;
import satkan.persistant.*;
```

```
public class DataBase {

    public DataBase() {
        PersistenceManager temppm = Utilities....
           getPersistenceManager(false);
        DataBase.pm = temppm;
    }

    public DataBase(Boolean clean) {
        PersistenceManager temppm = Utilities....
           getPersistenceManager(clean);
        DataBase.pm = temppm;
    }

    public void save_satellite(String satelliteId, ArrayList<...
       Momentable> componentList, ArrayList<double[]> ...
       rotationLocal, ArrayList<double[]> rotationGlobal, ...
       ArrayList<double[]> translation) {

        ArrayList<ComponentLocation> locationList = new ArrayList<...
           ComponentLocation>();
        for (int i = 0; i < componentList.size(); i++) {
            ComponentLocation location = new ComponentLocation("X"...
               , "CT@", translation.get(i), rotationGlobal.get(i),...
                rotationLocal.get(i));
            locationList.add(location);
        }

        Satellite sat = new Satellite(satelliteId, componentList, ...
           locationList);

        try {
            pm.currentTransaction().begin();
            pm.makePersistent(sat);
            pm.currentTransaction().commit();

            System.out.println("Satellite Saved...");
        } catch (RuntimeException x) {
            System.out.println("Error: " + x.getMessage());
        }
    }

    public void save_satellite(String satelliteId, ArrayList<...
       Momentable> componentList, ArrayList<ComponentLocation> ...
       locationList) {

        Satellite sat = new Satellite(satelliteId, componentList, ...
           locationList);

        try {
            pm.currentTransaction().begin();
            pm.makePersistent(sat);
            pm.currentTransaction().commit();

            System.out.println("Satellite Saved...");
        } catch (RuntimeException x) {
            System.out.println("Error: " + x.getMessage());
        }
    }

    public void delete_Data(String classname, String componentId) ...
        throws ClassNotFoundException {

        Class cls = Class.forName("satkan.persistant." + classname...
            );
        Query query = pm.newQuery(cls, "this.Id == componentId");
        query.declareParameters("String componentId");
        Collection result = (Collection) query.execute(componentId...
            );
        Iterator itr = result.iterator();
        Object obj = itr.next();
        //  System.out.println(obj.get_id());
        query.closeAll();
        pm.currentTransaction().begin();
        //  sobj.set_Id(obj.get_id());
        pm.deletePersistent(obj);
        pm.currentTransaction().commit();
    }

    public boolean has_Id(String classType, String componentId) ...
        throws ClassNotFoundException {

        Class cls = Class.forName("satkan.persistant." + classType...
            );
        Query query = pm.newQuery(cls, "this.Id == componentId");
        query.declareParameters("String componentId");
        Collection result = (Collection) query.execute(componentId...
            );
        Iterator itr = result.iterator();
        //      Object obj = itr.next();
        if (itr.hasNext()) {
            query.closeAll();
            return true;
        } else {
            query.closeAll();
        }
        return false;
    }

    public void ChangeOrientation(String classname, String ...
        componentId, String[] newType) throws ...
        ClassNotFoundException {
        //  this.pm = Utilities.getPersistenceManager(false);
        Class cls = Class.forName("satkan.persistant." + classname...
           );
        Query query = pm.newQuery(cls, "this.Id == componentId");
        query.declareParameters("String componentId");
        Collection result = (Collection) query.execute(componentId...
           );
        Iterator itr = result.iterator();

        Object obj = itr.next();

        if (obj.getClass().isInstance(new ComponentLocation())) {
            ComponentLocation objl = (ComponentLocation) obj;
            if (!objl.getConnectionType().equalsIgnoreCase("CTO"))...
                {
                System.out.println(objl.getId() + "Nothing changed...
                   ");
                query.closeAll();
            }
        } else {
            if (!obj.getClass().isInstance(new Plate())) {
                Component objc = (Component) obj;
                System.out.println(objc.getId());
                query.closeAll();
                pm.currentTransaction().begin();
                objc.set_orientation(newType);
                //   pm.deletePersistent(obj);
                pm.makePersistent(objc);
                pm.currentTransaction().commit();
            }
        }
    }

    public void ChangeConnectionType(String classname, String ...
       componentId, String newType) throws ClassNotFoundException ...
       {
        //  this.pm = Utilities.getPersistenceManager(false);
        Class cls = Class.forName("satkan.persistant." + classname...
           );
        Query query = pm.newQuery(cls, "this.Id == componentId");
        query.declareParameters("String componentId");
        Collection result = (Collection) query.execute(componentId...
           );
        Iterator itr = result.iterator();

        Object obj = itr.next();

        if (obj.getClass().isInstance(new ComponentLocation())) {
            ComponentLocation objl = (ComponentLocation) obj;
            if (!objl.getConnectionType().equalsIgnoreCase("CTO"))...
                {
                System.out.println(objl.getId());
                query.closeAll();
                pm.currentTransaction().begin();
                objl.set_connectionType(newType);
                //   pm.deletePersistent(obj);
                pm.makePersistent(objl);
                pm.currentTransaction().commit();
            }
        } else {
            if (!obj.getClass().isInstance(new Plate())) {
                Component objc = (Component) obj;
                System.out.println(objc.getId());
                query.closeAll();
                pm.currentTransaction().begin();
                objc.set_connectionType(newType);
                //   pm.deletePersistent(obj);
                pm.makePersistent(objc);
                pm.currentTransaction().commit();
            }
        }
    }

    public Object get_component_object(String classType, String ...
       componentId) throws ClassNotFoundException {

        Class cls = Class.forName("satkan.persistant." + classType...
           );
        Query query = pm.newQuery(cls, "this.Id == componentId");
        query.declareParameters("String componentId");
        Collection result = (Collection) query.execute(componentId...
           );
        Iterator itr = result.iterator();
        Object obj = itr.next();
        query.closeAll();
        return obj;
    }

    public Object get_component_object(String classType) throws ...
       ClassNotFoundException {

        Class cls = Class.forName("satkan.persistant." + classType...
           );
        Query query = pm.newQuery(cls);

        Collection result = (Collection) query.execute();
        Iterator itr = result.iterator();
        Object obj = itr.next();
        query.closeAll();
        return obj;
    }

    public String[] get_Id(String classType) throws ...
       ClassNotFoundException {
        Class cls = Class.forName("satkan.persistant." + classType...
            );
        Query query = pm.newQuery(cls);
        query.setOrdering("Id ascending");
        Collection results = (Collection) query.execute();
        Iterator itr = results.iterator();
        String[] strings = new String[results.size()];
        int i = 0;
        while (itr.hasNext()) {
            if (classType.equalsIgnoreCase("Satellite")) {
                Momentable obj = (Momentable) itr.next();
                strings[i] = obj.get_id();
                i++;
            } else {
                Component obj = (Component) itr.next();
                strings[i] = obj.getId();
                i++;
            }
        }
        query.closeAll();
        return strings;
    }

    public String[] get_Id(String classType, int availability) ...
        throws ClassNotFoundException {

        Class cls = Class.forName("satkan.persistant." + classType...
            );
        Query query = pm.newQuery(cls);
        query.setOrdering("Id ascending");
        Collection results = (Collection) query.execute();
        Iterator itr = results.iterator();
        ArrayList<String> alist = new ArrayList<String>();
        int j = 0;
        while (itr.hasNext()) {
            Component obj = (Component) itr.next();
            if (obj.getAvailability() >= availability) {
                alist.add(obj.getId());
            }
            j++;
        }
        String[] strings = new String[alist.size()];
        for (int i = 0; i < alist.size(); i++) {
            strings[i] = alist.get(i);
        }
        query.closeAll();
        return strings;
    }

    public String[] get_Id_ComponentLocations() throws ...
        ClassNotFoundException {
```

```
        Class cls = Class.forName("satkan.persistant....
           ComponentLocation");
        Query query = pm.newQuery(cls);
        //   query.setOrdering("Id");
        Collection results = (Collection) query.execute();
        Iterator itr = results.iterator();
        String[] strings = new String[results.size()];
        int loci = 0;
        while (itr.hasNext()) {
            ComponentLocation obj = (ComponentLocation) itr.next()...
               ;
            strings[loci] = obj.getId();
            loci++;
        }

        query.closeAll();
        return strings;
    }

    public int get_Component_num(String strn) throws ...
       ClassNotFoundException {
        int totalnumber = 0;
        String[] str = this.get_Id(strn);
        for (int i = 0; i < str.length; i++) {
            Component comp = (Component) this.get_component_object...
               (strn, str[i]);
            totalnumber = totalnumber + comp.getAvailability();
        }
        return totalnumber;
    }

    public String[] get_Availability(String classType) throws ...
       ClassNotFoundException {

        Class cls = Class.forName("satkan.persistant." + classType...
           );
        Query query = pm.newQuery(cls);
        query.setOrdering("Id ascending");
        Collection results = (Collection) query.execute();
        Iterator itr = results.iterator();
        String[] strings = new String[results.size()];
        int i = 0;
        while (itr.hasNext()) {
            Component obj = (Component) itr.next();
            strings[i] = String.valueOf(obj.getAvailability());
            i++;
        }
        query.closeAll();
        return strings;
    }
```

```
public String[] get_Id(String classType, String field1, String...
    field1Value, String field2, String field2Value) throws ...
   ClassNotFoundException {

    Class cls = Class.forName("satkan.persistant." + classType...
       );
    Query query = pm.newQuery(cls, "this." + field1 + " == ...
       field1Value &&  this." + field2 + " == field2Value");
    query.declareParameters("String field1Value,String ...
       field2Value");
    Collection result = (Collection) query.execute(field1Value...
       ,field2Value);
    Iterator itr = result.iterator();

    String[] strings = new String[result.size()];
    int i = 0;
    while (itr.hasNext()) {
        ComponentLocation obj = (ComponentLocation) itr.next()...
           ;
        strings[i] = obj.getId();
        i++;
    }
    query.closeAll();
    return strings;
}

public ArrayList<String[]> get_ConnectionTypes() throws ...
   ClassNotFoundException {

    Class cls = Class.forName("satkan.persistant....
       ComponentLocation");
    Query query = pm.newQuery(cls);
    query.setOrdering("connectionType ascending");
    Collection results = (Collection) query.execute();
    Iterator itr = results.iterator();
    int i = 0;
    String type = "";
    int count = 0;
    ArrayList<String[]> typeCount = new ArrayList<String[]>();

    while (itr.hasNext()) {
        ComponentLocation obj = (ComponentLocation) itr.next()...
           ;
        System.out.println(obj.getConnectionType());
        if (i == 0) {
            type = obj.getConnectionType();
            count++;
            i++;
        } else {
            if (obj.getConnectionType().equals(type)) {
                count++;
            } else {
```

```
                        String[] connectionType = new String[2];
                        connectionType[0] = type;
                        connectionType[1] = String.valueOf(count);
                        typeCount.add(connectionType);
                        type = obj.getConnectionType();
                        count = 1;
                    }
                }
            }
            String[] connectionType = new String[2];
            connectionType[0] = type;
            connectionType[1] = String.valueOf(count);
            typeCount.add(connectionType);

            query.closeAll();
            return typeCount;
        }
        private static PersistenceManager pm;

        public void setAvailability(String componentId, int avai) ...
           throws ClassNotFoundException {
            //  this.pm = Utilities.getPersistenceManager(false);
            Class cls = Class.forName("satkan.persistant.Component");
            Query query = pm.newQuery(cls, "this.Id == componentId");
            query.declareParameters("String componentId");
            Collection result = (Collection) query.execute(componentId...
               );
            Iterator itr = result.iterator();
            Object obj = itr.next();
            Component objc = (Component) obj;

            query.closeAll();
            pm.currentTransaction().begin();
            objc.set_availabilit(avai);
            pm.makePersistent(objc);
            pm.currentTransaction().commit();
        }
    }
```

Listing A.23: Calculate Inertia Class

```
package satkan;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.vecmath.Matrix3d;
/**
 * Calculates the Moment of inertia at the center of a assemly
 * based on given properties, rotations and translation data
 * of each components
 *
 * */
public class CalculateInertia {

    public CalculateInertia(ArrayList<Momentable> complist, ...
        ArrayList<double[]> rotationLocal, ArrayList<double[]> ...
        rotationGlobal, ArrayList<double[]> translation) {
         this.list = complist;
         this.listRtl = rotationLocal;
         this.listRtg = rotationGlobal;
         this.listTr = translation;
    }

    public CalculateInertia() {

         list = new ArrayList<Momentable>();
         sum_mass = 0;
         center_of_mass[0] = 0;
         center_of_mass[1] = 0;
         center_of_mass[2] = 0;
         firstMoments[0] = 0;
         firstMoments[1] = 0;
         firstMoments[2] = 0;
         //set default values
         for (int i = 0; i < 9; i++) {
             inertia_xyz[i] = 0;
         }
    }
    public double sum_mass;
    private double[] center_of_mass = new double[3];
    private double[] firstMoments = new double[3];
    //Distance between  the center of mass of satellite from ...
        center of mass of each n components
    private double[][] delta_xyz;
    // Moment of inertia Ixx ,Iyy, Izz,Ixy,Ixz,Iyz
    private double[] inertia_xyz = new double[9];
    private ArrayList<Momentable> list;
    private ArrayList<double[]> listRtl;
    private ArrayList<double[]> listRtg;
    private ArrayList<double[]> listTr;

    public void add(Momentable obj) {
         list.add(obj);
    }

    // Calculate the total mass of the assembly
    public double get_sum_mass() {
         sum_mass = 0;
         for (Momentable a : list) {
             sum_mass = sum_mass + a.get_mass();
         }
         return sum_mass;
    }
```

```

//sum of moments to find center of mass, by dividing it by ...
    total mass of the system
public void sumfirstMoments() throws ClassNotFoundException {
    firstMoments[0] = 0;
    firstMoments[1] = 0;
    firstMoments[2] = 0;
    for (int j = 0; j < list.size(); j++) {
        Momentable a = list.get(j);
        double[] arrayTr = this.get_translation(j);
        double[] arrayRt = this.get_rotation_global(j);
        //convert center of mass into matrix form and multiply...
            with global  rotation matrix
        Matrix3d mat1 = new Matrix3d();
        Matrix3d mat2 = new Matrix3d();
        mat1.set(arrayRt);
        mat2.setRow(0, a.get_cg());
        mat2.mul(mat1);
        //create a variable for rotated center of mass ...
            coordinates
        //define variable for rotated ceneter of gravity
        double[] cog = new double[3];
        mat2.getRow(0, cog);
        System.out.println("rotated center of mass \n" + ...
            Arrays.toString(cog));
        for (int i = 0; i < firstMoments.length; i++) {
            firstMoments[i] = firstMoments[i] + (arrayTr[i] + ...
                cog[i]) * a.get_mass();
        }
    }
}

public double[] get_center_of_mass() throws ...
    ClassNotFoundException {
    this.sumfirstMoments();
    for (int i = 0; i < center_of_mass.length; i++) {
        center_of_mass[i] = firstMoments[i] / this....
            get_sum_mass();
    }
    return center_of_mass;
}

// distance between the center of the system and translated ...
    center of  individual component
public void delta_xyz() throws ClassNotFoundException {
    delta_xyz = new double[list.size()][3];
    for (int i = 0; i < list.size(); i++) {
        Momentable p = list.get(i);
        //get translation values from Enum Translation
        double[] arrayTr = new double[3];
        arrayTr = this.get_translation(i);
        double[] arrayRt = new double[9];
```

```
            arrayRt = this.get_rotation_global(i);
            System.out.println("arrayRT \n" + Arrays.toString(...
                arrayRt));
            //convert center of mass in a matrix form and multiply...
                 with rotation matrix
            Matrix3d mat1 = new Matrix3d();
            Matrix3d mat2 = new Matrix3d();
            mat1.set(arrayRt);
            mat2.setRow(0, p.get_cg());
            mat2.mul(mat1);
            //create a variable for rotated center of mass ...
                coordinates
            double[] cog = new double[3];
            mat2.getRow(0, cog);
            System.out.println("rotated center of mass \n" + ...
                Arrays.toString(cog));

            for (int j = 0; j < 3; j++) {
                delta_xyz[i][j] = arrayTr[j] + cog[j] - ...
                    get_center_of_mass()[j];
            }
        }
    }

    public double[] get_inertia_xyz() throws ...
        ClassNotFoundException {
        delta_xyz();
        for (int i = 0; i < 9; i++) {
            inertia_xyz[i] = 0;
        }

        for (int i = 0; i < list.size(); i++) {
            Momentable p = list.get(i);

            Matrix3d localI = new Matrix3d();
            double[] localInertiaArray = new double[3];
            localInertiaArray = p.get_localInertiaTensor();
            localI.setM00(localInertiaArray[0]);
            localI.setM11(localInertiaArray[1]);
            localI.setM22(localInertiaArray[2]);

            Matrix3d R = new Matrix3d();
            Matrix3d Rtrans = new Matrix3d();
            double[] RotationArray = new double[9];
            //use local rotation matrix to calculate local Inertia...
                 Tensor
            RotationArray = this.get_rotation_local(i);
            //apply rotation matrix
            R.set(RotationArray);
            Rtrans.transpose(R);
            localI.mul(Rtrans);
            localI.mul(R, localI);
            inertia_xyz[0] = inertia_xyz[0] + localI.m00 + p....
               get_mass() * (Math.pow(delta_xyz[i][1], 2) + Math....
               pow(delta_xyz[i][2], 2));
            inertia_xyz[4] = inertia_xyz[4] + localI.m11 + p....
               get_mass() * (Math.pow(delta_xyz[i][0], 2) + Math....
               pow(delta_xyz[i][2], 2));
            inertia_xyz[8] = inertia_xyz[8] + localI.m22 + p....
               get_mass() * (Math.pow(delta_xyz[i][0], 2) + Math....
               pow(delta_xyz[i][1], 2));
            inertia_xyz[1] = inertia_xyz[1] + delta_xyz[i][0] * ...
               delta_xyz[i][1] * p.get_mass();
            inertia_xyz[3] = inertia_xyz[3] + delta_xyz[i][0] * ...
               delta_xyz[i][1] * p.get_mass();
            inertia_xyz[2] = inertia_xyz[2] + delta_xyz[i][0] * ...
               delta_xyz[i][2] * p.get_mass();
            inertia_xyz[6] = inertia_xyz[6] + delta_xyz[i][0] * ...
               delta_xyz[i][2] * p.get_mass();
            inertia_xyz[5] = inertia_xyz[5] + delta_xyz[i][1] * ...
               delta_xyz[i][2] * p.get_mass();
            inertia_xyz[7] = inertia_xyz[7] + delta_xyz[i][1] * ...
               delta_xyz[i][2] * p.get_mass();
        }
        return inertia_xyz;
    }
    public String toString() {
        try {
            String info;
            double[] inertia = new double[9];
            inertia = this.get_inertia_xyz();
            DecimalFormat df = new DecimalFormat("#0.0000000");
            info = "Moment of inertia " + "\n" + df.format(inertia...
               [0]) + "   " + df.format(inertia[1]) + "   " + df....
               format(inertia[2]) + "\n" + df.format(inertia[3]) +...
                "   " + df.format(inertia[4]) + "   " + df.format(...
               inertia[5]) + "\n" + df.format(inertia[6]) + "   " +...
                df.format(inertia[7]) + "   " + df.format(inertia...
               [8]) + "\nCenter of Mass: " + "\n" + df.format(this...
               .get_center_of_mass()[0]) + "   " + df.format(this....
               get_center_of_mass()[1]) + "  " + df.format(this....
               get_center_of_mass()[2]) + "\nTotal mass: " + "\n" ...
               + df.format(this.get_sum_mass());
            return info;
        } catch (ClassNotFoundException ex) {
            Logger.getLogger(CalculateInertia.class.getName()).log...
               (Level.SEVERE, null, ex);
            return "";
        }
    }

    public double[] get_translation(int listIndex) {
        return listTr.get(listIndex);
    }

    public double[] get_rotation_global(int listIndex) {

        return listRtg.get(listIndex);
    }

    public double[] get_rotation_local(int listIndex) {

        return listRtl.get(listIndex);
    }
}
```

Bibliography

1. Arritt B. J., Buckley S.J. and Ganley J.M. “Development of a Satellite Structural Architecture for Operationally Responsive Space”. *SPIE Smart Structures and Materials and NDE*. Society of Photo-optical Instrumentation Engineers, San Diego, California, March 2008.

2. Arritt B. J., Buckley S.J. and Ganley J.M. “Structural Health Monitoring: an Enabler for Responsive Satellites”. *SPIE Smart Structures and Materials and NDE*. Society of Photo-optical Instrumentation Engineers, San Diego, California, March 2008.

3. Board, IEEE Standards. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers, New York, 1st edition, 1990.

4. Commons, Wikimedia. *Normal Distribution PDF*. internet: commons.wikimedia.org, May 2008.

5. Deitel H. M., Deitel P. J. *Java How to Program,*. Prentice Hall, 2nd edition, 2004.

6. Ferebee M.J., Monell D.W. and Troutman P.A. “Satellite Systems Design/Simulation Environment - A Systems Approach To Pre-Phase A Design”. 35th AIAA Aerospace Sciences Meeting and Exhibition, January 1997.

7. From J., Kramer S. and Pohl E. “A Small Satellite System Design Process”. *Aerospace and Electronics Conference*, 1(2):423–429, July 1997.

8. Fronterhouse D., Lyke J. and Achramowicz S. “Plug-and-play Satellite (PnPSat)”. AIAA conferance, January 2007.

9. Gan K. K., Kagan H. P. and Kass R. D. “Simple Demonstration Of The Central Limit Theorem Using Mass Measurements”, September 2001.

10. Gray A., Abbena E. *Modern Differential Geometry of Curves and Surfaces with Mathematica*. CRC Press, New York, 2nd edition, 2002.

11. Knight, D. “Concept of Operations for Operationally Responsive Space”. American Institute of Aeronautics and Astronautics, April 2006.

12. Lanza D., Lyke J. et al. “Responsive Space Through Adaptive Avionics”. *Space 2004 Conference and Exhibit*. American Institute of Aeronautics and Astronautics, San Diego, California, September 2004.

13. Petter, S. *Standard Deviation Diagram*. internet: en.wikipedia.org, May 2008.

14. Puhlhofer T., Langer H. et al. "Multicriteria And Discrete Configuration And Design Optimization With Applications For Satellites". *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. American Institute of Aeronautics and Astronautics, Albany, New York, August 2004.

15. Puhlhofer T., Langer H. et al. "Space Plug-And-Play Avionics". *3rd Responsive Space Conference*. American Institute of Aeronautics and Astronautics, Los Angeles, California, April 2005.

16. Santangelo, A. "OpenSAT and SATBuilder: A Satellite Design Automation Environment for Responsive Space". 46th AIAA Aerospace Sciences Meeting and Exhibition, January 2008.

17. Sierra K., Bates B. *Head First Java*. O'reilly, 2nd edition, 2005.

18. Team, ObjectDB Developer. *ObjectDB Developer's Guide*. internet: www.objectdb.com, May 2008.

19. Trunce R., Eddy C. "Responsive Space's Spacecraft Design Tool (SDT)". *4rd Responsive Space Conference*. American Institute of Aeronautics and Astronautics, Los Angeles, California, April 2006.

20. Turner, A.J. *An Open-Source, Extensible Spacecraft Simulation And Modeling Environment Framework*. Master's thesis, Virginia Polytechnic Institute and State University, 2003.

21. Wertz, J.R. "The Need for Responsive Space". 3rd Responsive Space Conference, April 2005.

22. Wertz J. R., Larson W. J. *Space Mission Analysis and Design*. Springer - Verlag, 3rd edition, 1999.

23. Wertz J. R., Larson W. J. *Reducing Space Mission Cost*. Microcosm Press, 3rd edition, 2005.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 074-0188*

1. REPORT DATE *(DD-MM-YYYY)* 11 September 2008	2. REPORT TYPE **Thesis**	3. DATES COVERED *(From – To)* 05/01/2007-09/11/2008
4. TITLE AND SUBTITLE A Constraint Based Approach for Building Operationally Responsive Satellites		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Mesut Ozkan KAHRAMAN		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology, Graduate School of Engineering and Management (AFIT/ENY), 2950 Hobson Way, WPAFB OH 45433-7765		8. PERFORMING ORGANIZATION REPORT NUMBER **AFIT/GSS/ENY/08-S02**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Operational Responsive Space (ORS) program requires flexible and responsive satellites to meet user's needs. Traditional satellite design methods are typically iterative processes that optimize individual components, subsystems, and ultimately the entire satellite. This study focuses on developing a new approach for creating Responsive Satellites (RS) from Plug-and-Play (PnP) components. The aim is to create an approach that quickly evaluates a wide variety of possible satellite configurations and identify the best configurations that meet the users' needs and constraints.

Satellite configurations are created by matching locations on the satellite structure with PnP components. Various constraints are derived from the user's inputs at different levels of the configuration process. As the user provides more information related to PnP satellite, additional constraints can be applied to reduce the number of PnP satellite configurations resulting in manageable numbers or even zero configurations.

In this research, we found that applying constraints whenever it is applicable results in eliminating invalid configurations. Each satellite configuration is saved to a database, if the user desires, a sorted list can help the user find the lowest mass and least expensive satellite that meets their requirements. Configurations can also be eliminated when respective properties are very close to each other which will reduce the number of satellite configurations from which the user can select. A goal of this research effort is to help users assess basic concept feasibility from several key aspects in a short period of time. Finally, more specialized and computationally demanding estimation tools could be called from this approach to perform further analysis, such as thermal, vibration, or structural, to compare and contrast performance characteristics of various satellite configurations.

15. SUBJECT TERMS
Responsive, Satellite, Constraint Based Approach, Design, Configuration

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT					Advisor Name Eric D Swenson
U	U	U	UU	145	19b. TELEPHONE NUMBER *(Include area code)* Advisor phone and e-mail (937)-255-3636 extension 7479 eric.swenson@afit.edu